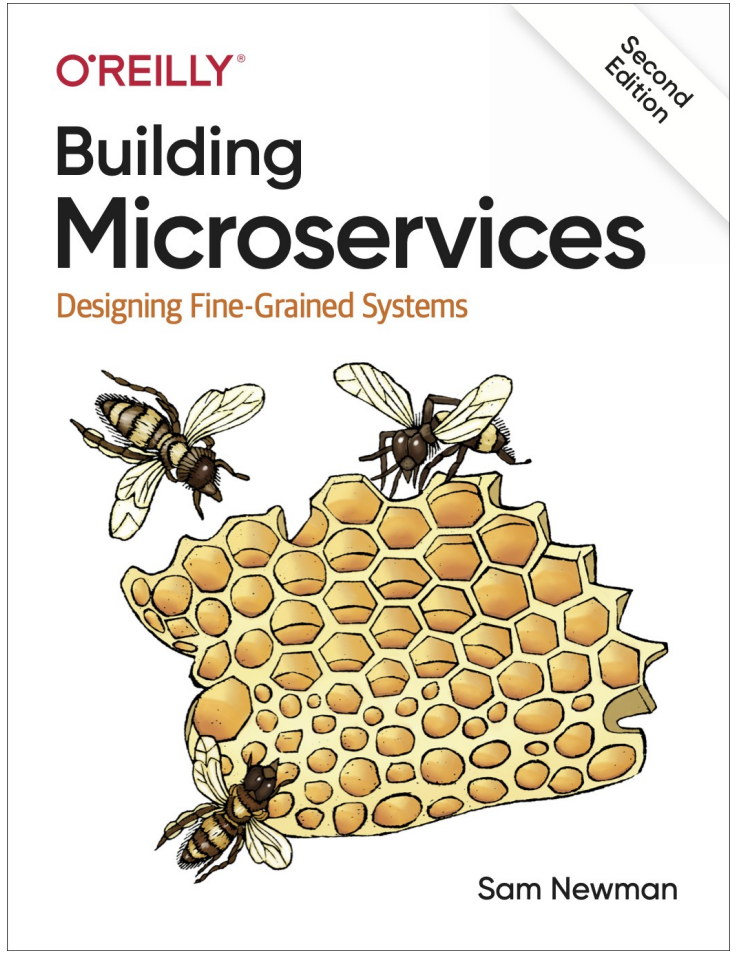
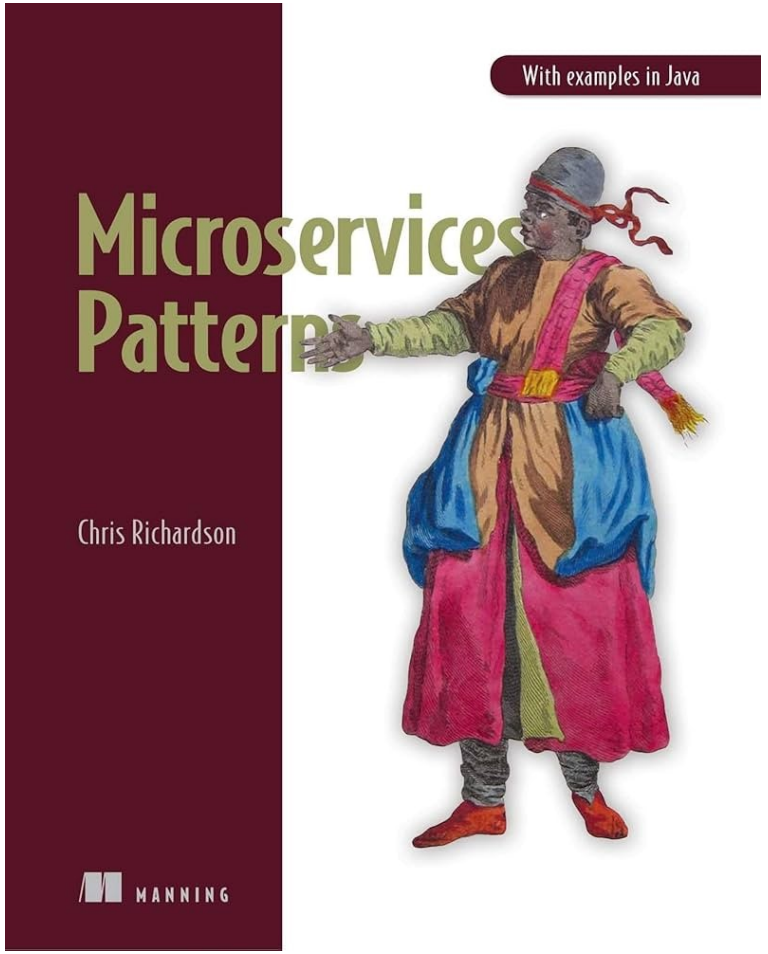
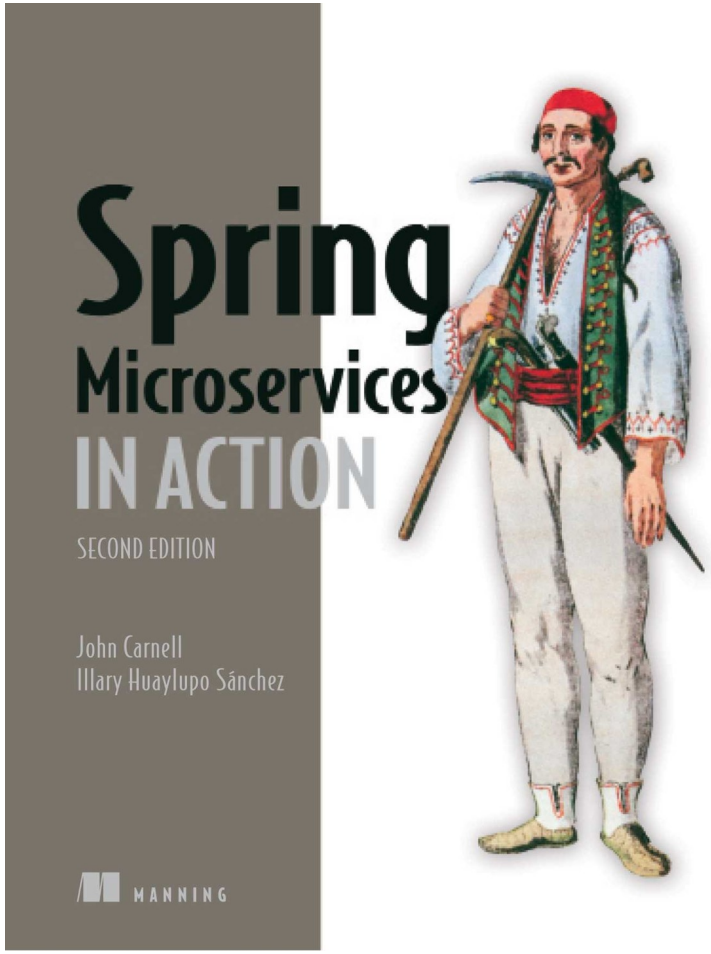
Arquitecturas Basadas en Servicios (Microservicios)

“El diseño arquitectónico de software es la clave para construir sistemas sostenibles.”

— Grady Booch


O'REILLY
Second Edition
Building
Microservices
Designing Fine-Grained Systems
Sam Newman

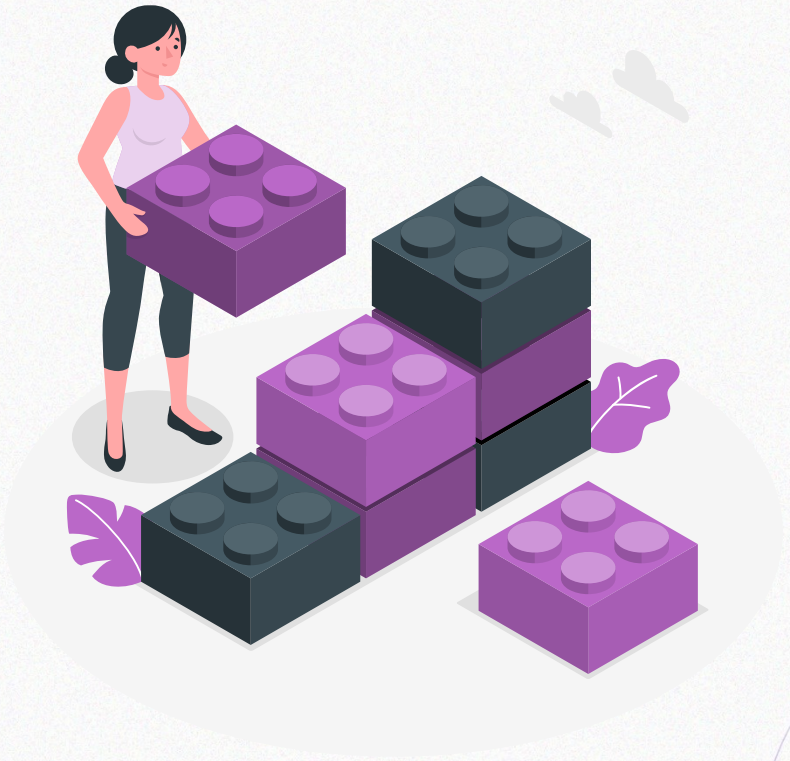

With examples in Java
Microservices
Patterns
Chris Richardson
MANNING


Spring
Microservices
IN ACTION
SECOND EDITION
John Carnell
Illary Huaylupo Sánchez
MANNING

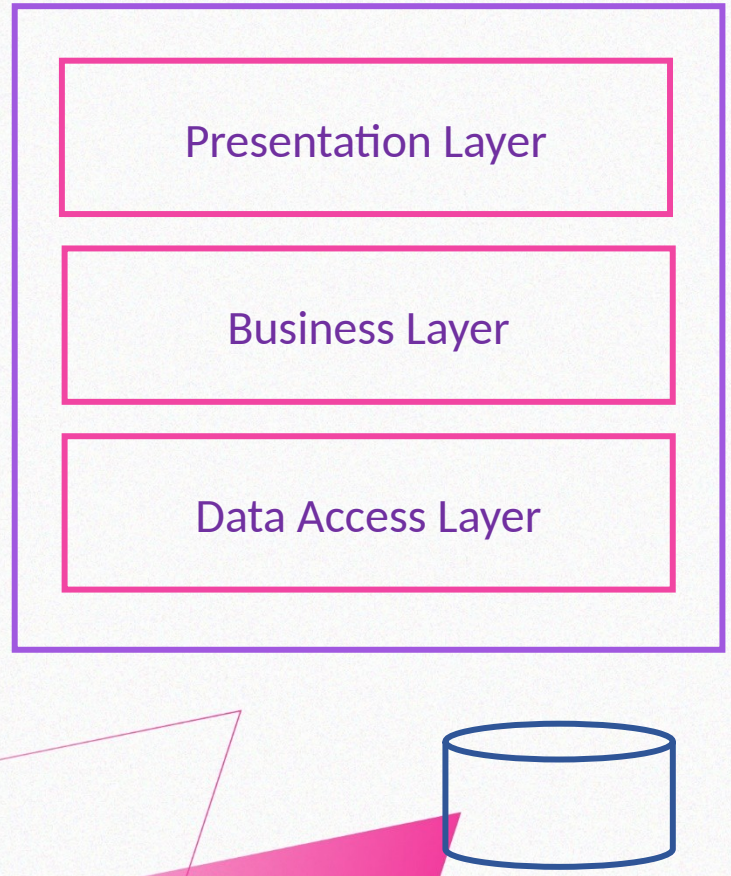
Diferencias entre Monolitos y Microservicios

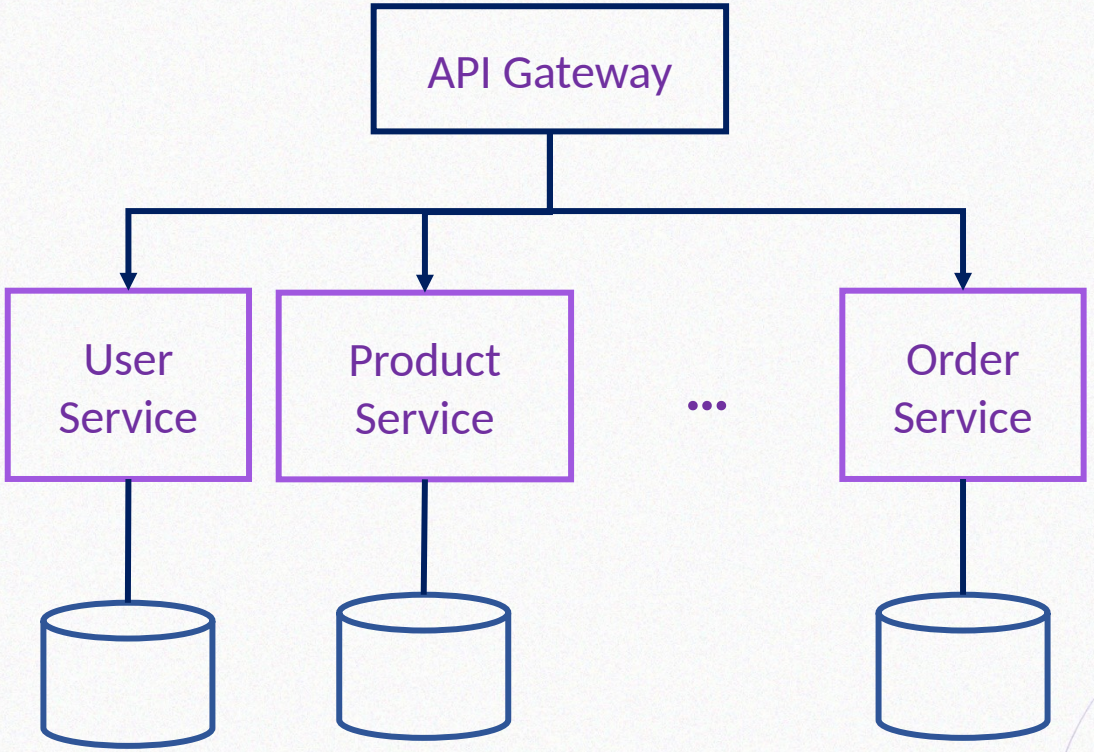
Monolito: Una arquitectura monolítica es aquella en la que toda la funcionalidad de una aplicación está contenida en una única unidad desplegable.

Microservicios: Los microservicios son un estilo arquitectónico donde la aplicación se descompone en pequeños servicios independientes, cada uno ejecutando una funcionalidad específica del negocio.



Diferencias entre Monolitos y Microservicios

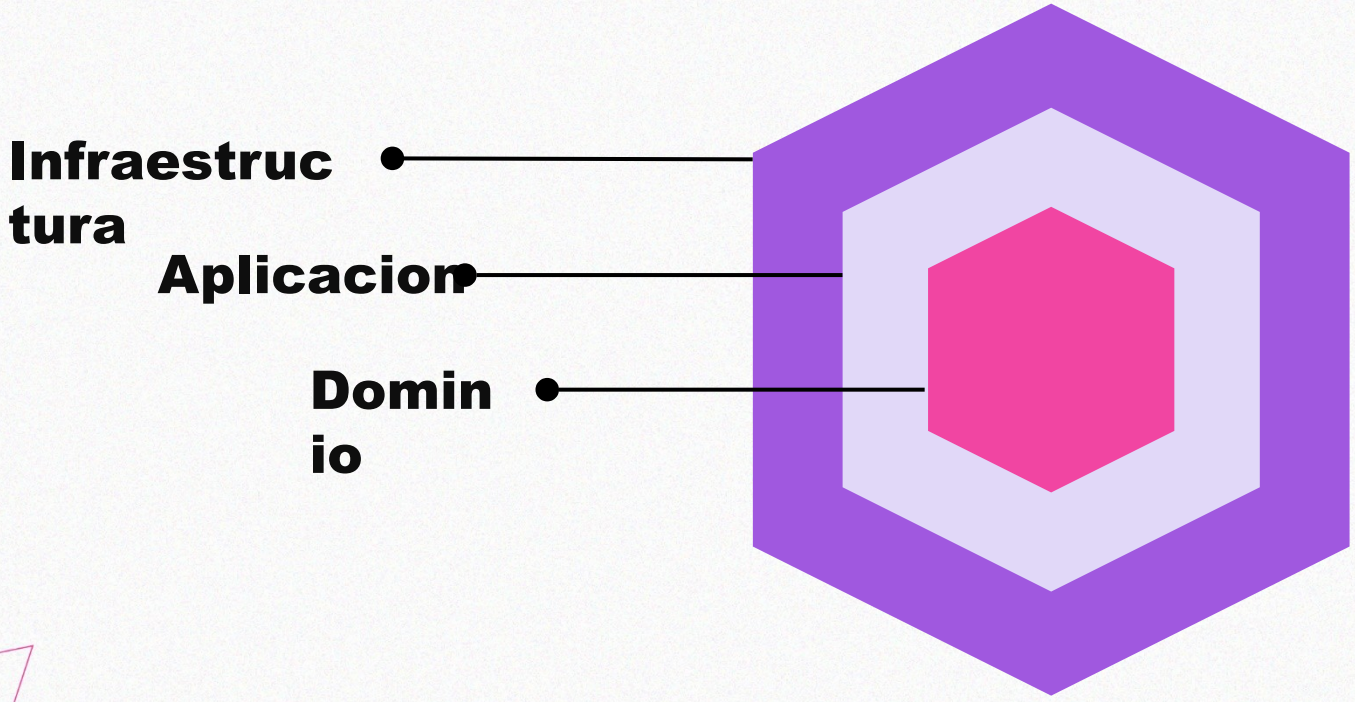




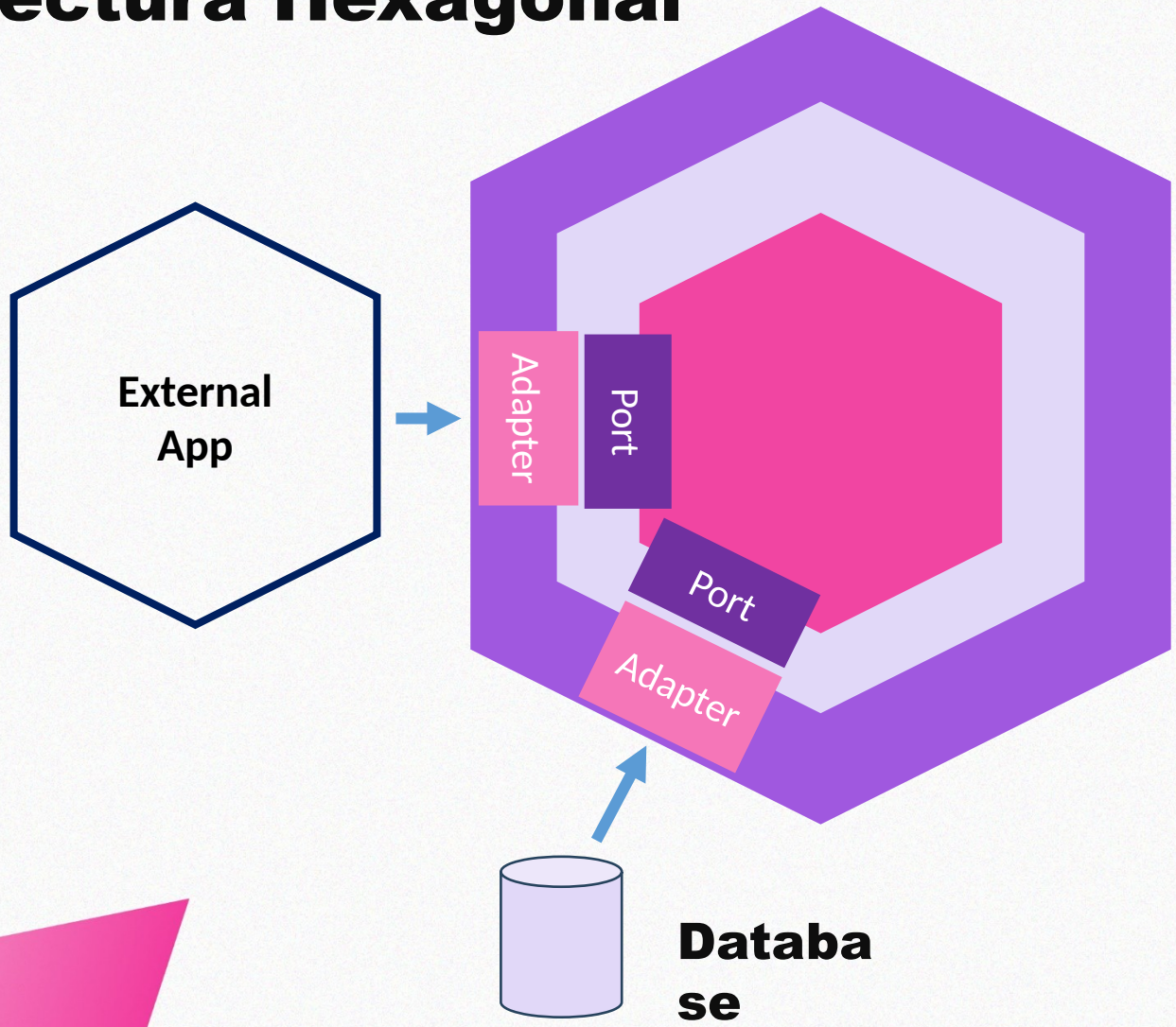
Aspecto	Monolito	Microservicios
Arquitectura	Una sola unidad desplegable.	Múltiples servicios independientes.
Tamaño del Equipo	Ideal para equipos pequeños.	Compatible con equipos distribuidos.
Escalabilidad	Escalabilidad horizontal limitada.	Escalabilidad independiente por servicio.
Mantenimiento	Cambios afectan todo el sistema.	Cambios limitados al servicio afectado.
Despliegue	Todo el sistema debe redeployarse.	Servicios se despliegan individualmente.
Desempeño	Mejor desempeño inicial por comunicación interna.	Penalización por comunicación a través de red.
Gestión de Datos	Base de datos única.	Base de datos independiente por servicio.
Complejidad	Menos compleja inicialmente.	Complejidad alta en gestión de servicios.

Patrones de Composición

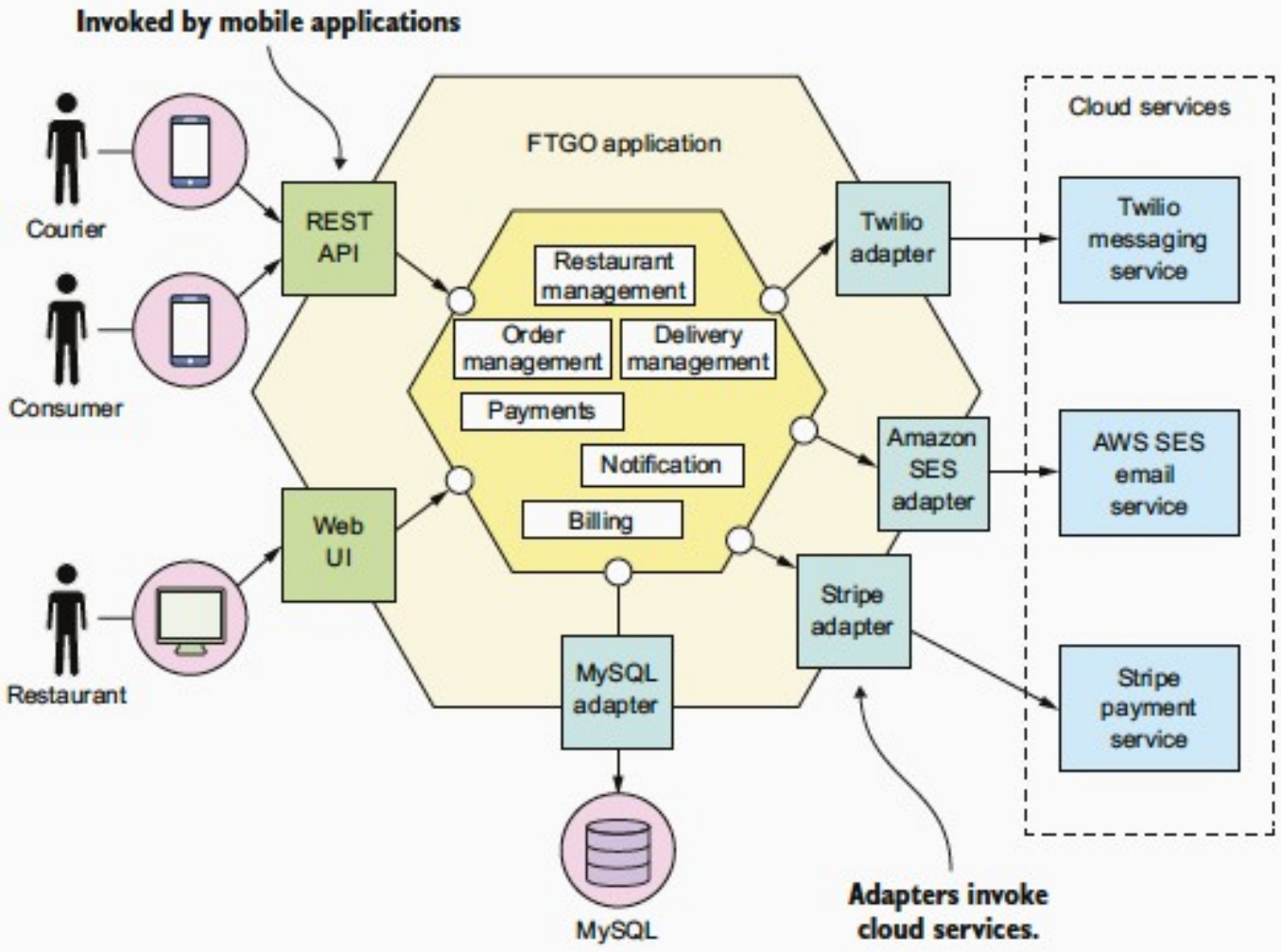
Arquitectura Hexagonal



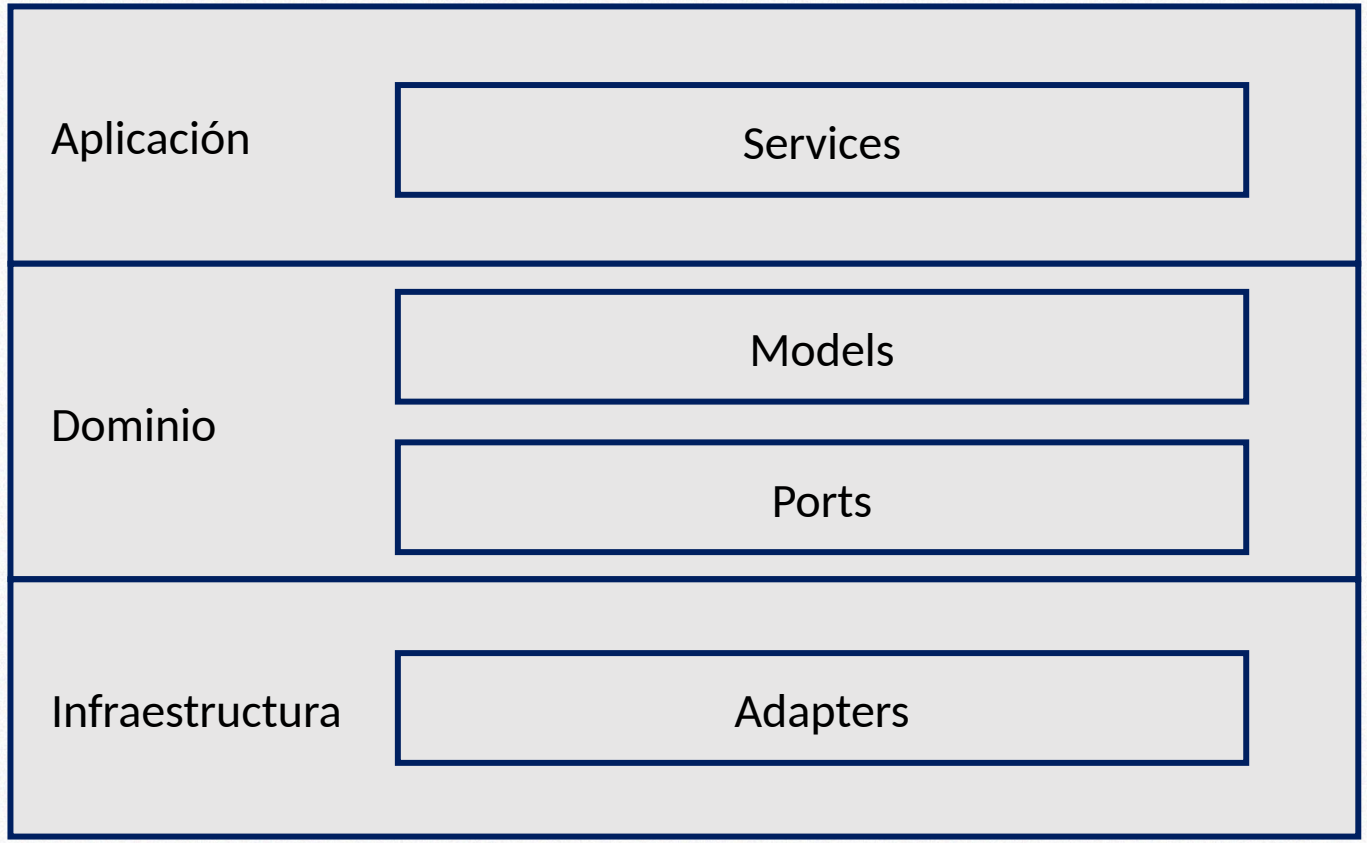
Arquitectura Hexagonal



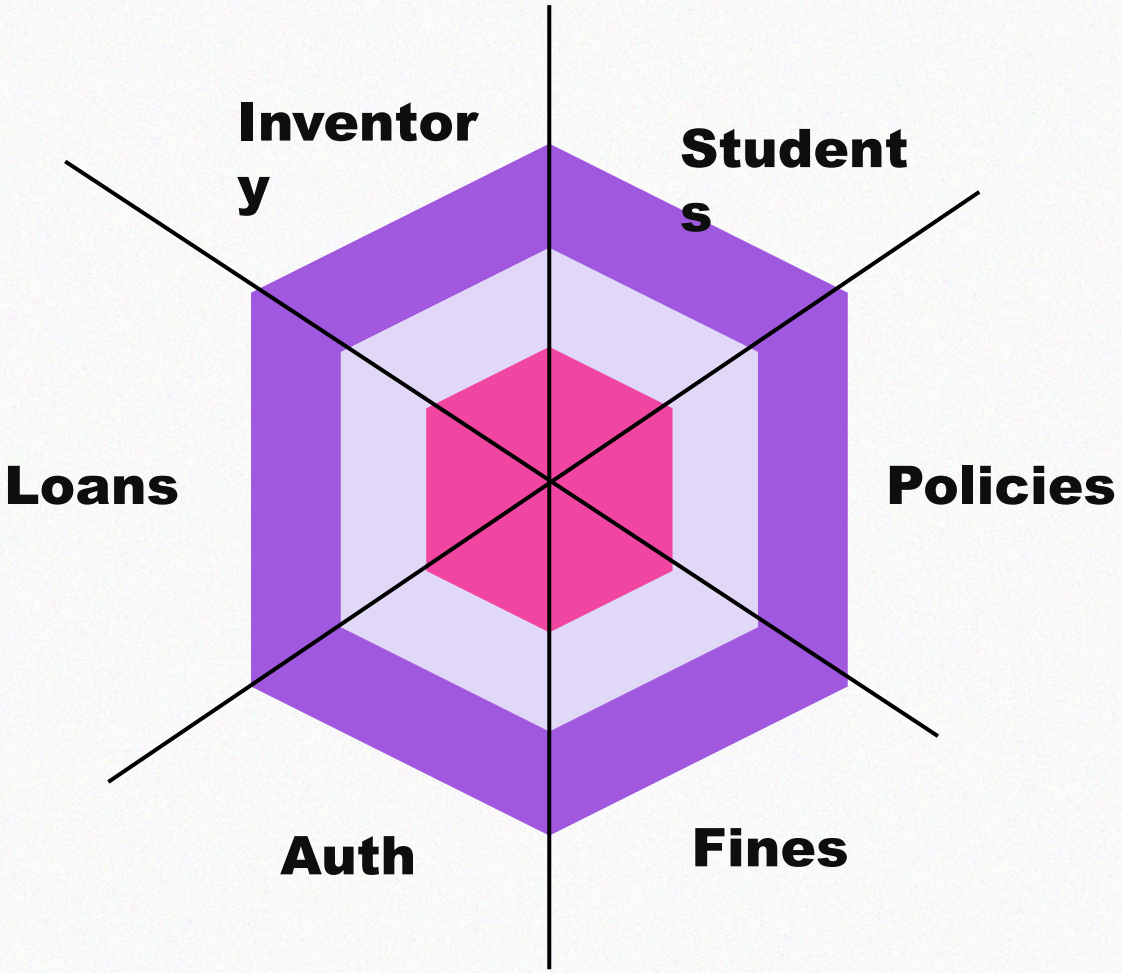
Arquitectura Hexagonal



Arquitectura Hexagonal

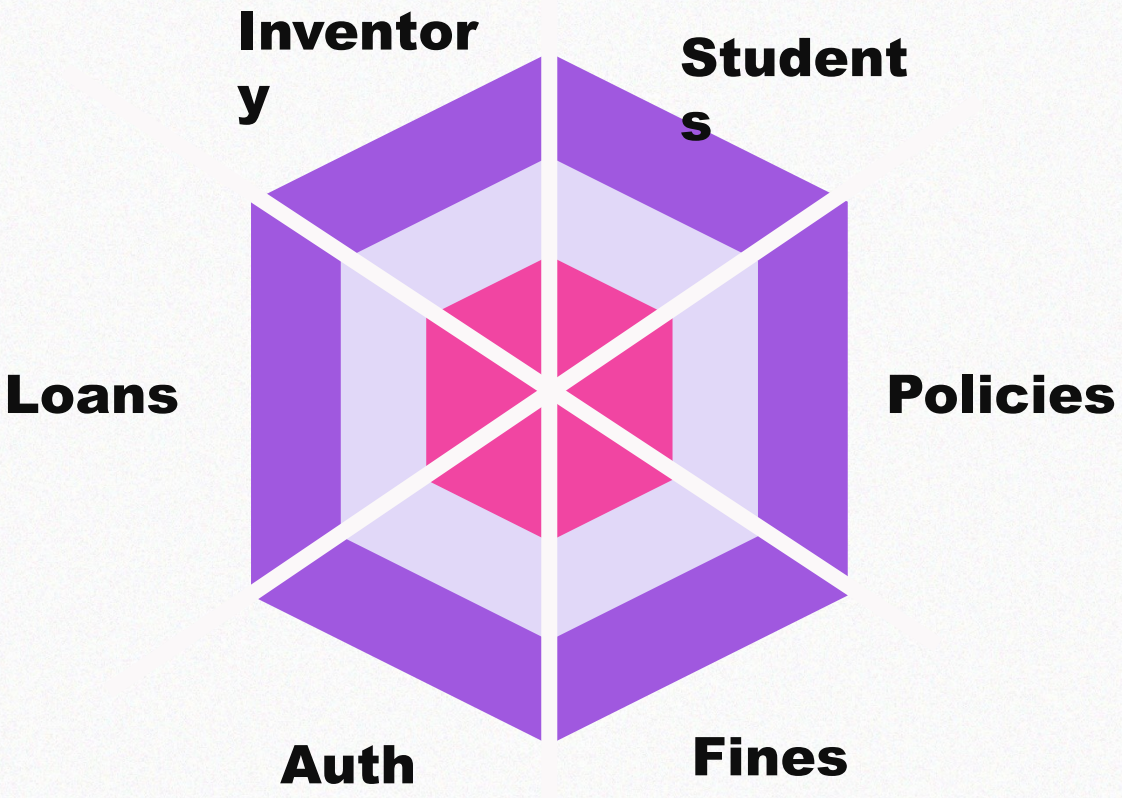


Arquitectura Hexagonal

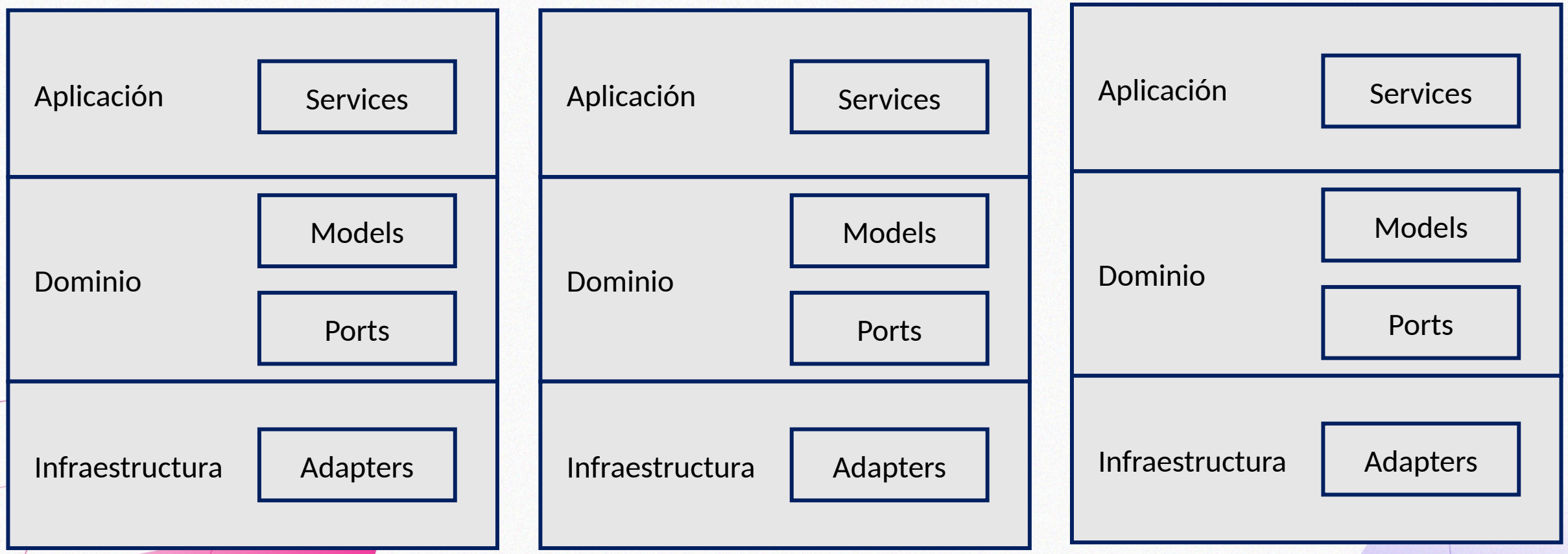


Arquitectura Hexagonal

Vertical Slicing



Arquitectura Hexagonal

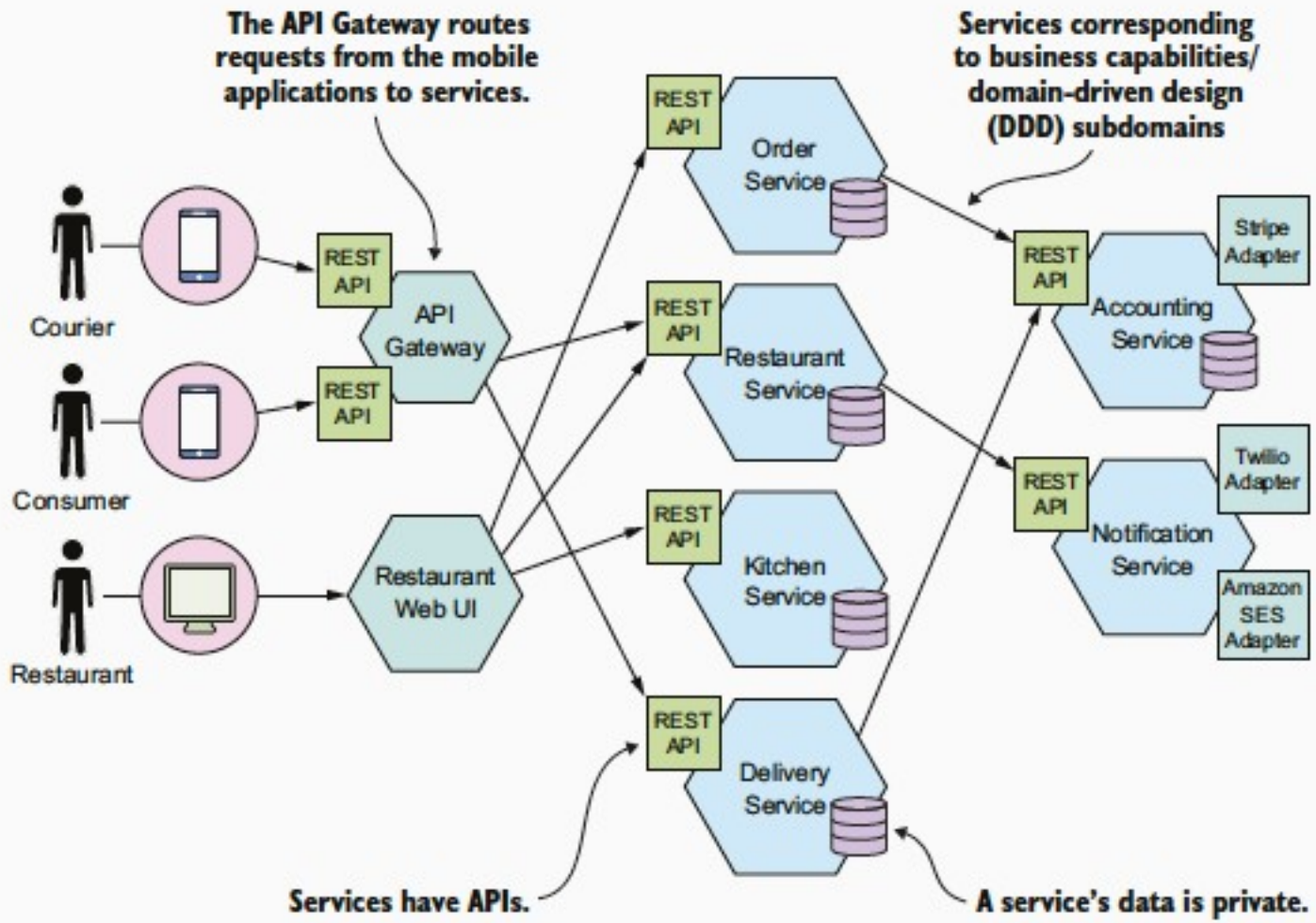


Inventor y

Loans

Student s

Arquitectura Hexagonal



Arquitectura Hexagonal

Dominio

```java
public class Product {
    private String id;
    private String name;
    private double price;

    // Constructor, getters y setters
    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Lógica de negocio
    public void applyDiscount(double percentage) {
        this.price -= this.price * (percentage / 100);
    }
}
```

Arquitectura Hexagonal

Dominio

```java
public class Product {
    private String id;
    private String name;
    private double price;

    // Constructor, getters y setters
    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    // Lógica de negocio
    public void applyDiscount(double percentage) {
        this.price -= this.price * (percentage / 100);
    }
}
```

Arquitectura Hexagonal

Puerto

```
public interface ProductRepository {
    void save(Product product);
    Product findById(String id);
}
```

Arquitectura Hexagonal

Servicio

```java
public class ProductService {
    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public void createProduct(String id, String name, double price) {
        Product product = new Product(id, name, price);
        productRepository.save(product);
    }

    public Product getProduct(String id) {
        return productRepository.findById(id);
    }
}
```

Arquitectura Hexagonal

Adaptador

```java
import java.util.HashMap;
import java.util.Map;


public class MySQLProductRepository implements ProductRepository {
    private Map<String, Product> database = new HashMap<>();

    @Override
    public void save(Product product) {
        database.put(product.getId(), product);
    }

    @Override
    public Product findById(String id) {
        return database.get(id);
    }
}
```

Domain Model

¿Qué es un Domain Model?

El **Domain Model** es una abstracción del dominio del negocio, representado mediante:

- **Entidades**: Objetos con identidad única dentro del dominio.
- **Value Objects**: Objetos que representan valores (inmutables) y no tienen identidad propia.
- **Agregados**: Unidades de consistencia que agrupan entidades relacionadas.
- **Servicios de Dominio**: Operaciones del dominio que no encajan en una sola entidad.
- **Eventos de Dominio**: Representan cambios o acciones significativas que ocurren dentro del dominio.

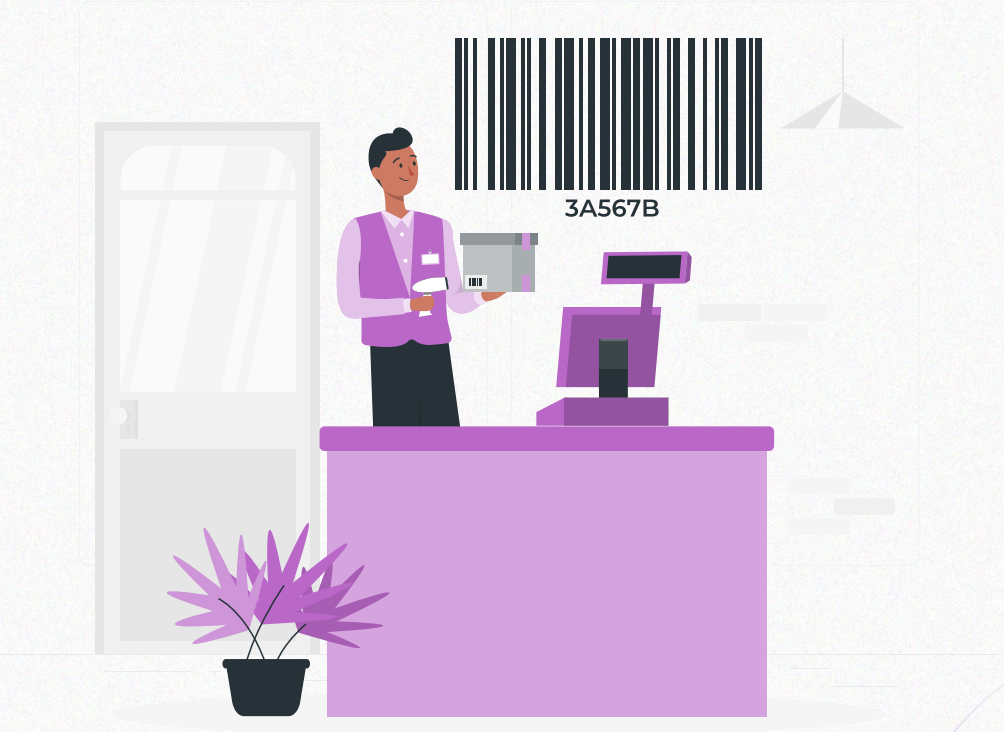
Entidades

Una **Entidad** es un objeto del modelo que tiene **identidad única**, independientemente de sus atributos.

Ejemplo
:

• En una biblioteca: un `Estudiante` con un ID único (como matrícula).

• Otro ejemplo: un `Libro` identificado por su ISBN.



Cómo reconocerlas:

¿Necesitas identificar este objeto incluso si sus atributos cambian? Si la respuesta es “sí”, probablemente sea una entidad.

¿Esto es una entidad?

- **Estudiante**
- **Multa**
- **Prestamo**

Value Objects

Un **Objeto de Valor** representa un concepto que se identifica por sus **atributos** en lugar de su identidad.

Ejemplo:

- **`Fecha de Devolución`:** No importa “qué” instancia sea, lo relevante es la fecha como valor.
- **`Dinero`:** Un monto con una moneda específica.

Cómo reconocerlas:

¿El objeto es inmutable y solo representa un valor o conjunto de valores? Si “sí”, es un Value Object.

¿Esto es un Value Object?

- **Libro**
- **Ultimo Prestatario**
- **Fecha de Prestamo**

Agregados

Un **Agregado** es un grupo de entidades y objetos de valor que se consideran una unidad de consistencia.

Ejemplo:

- **Raíz del agregado:** `Préstamo`.
- **Entidades relacionadas:** `Libro` y `Estudiante`.

Cómo reconocerlas:

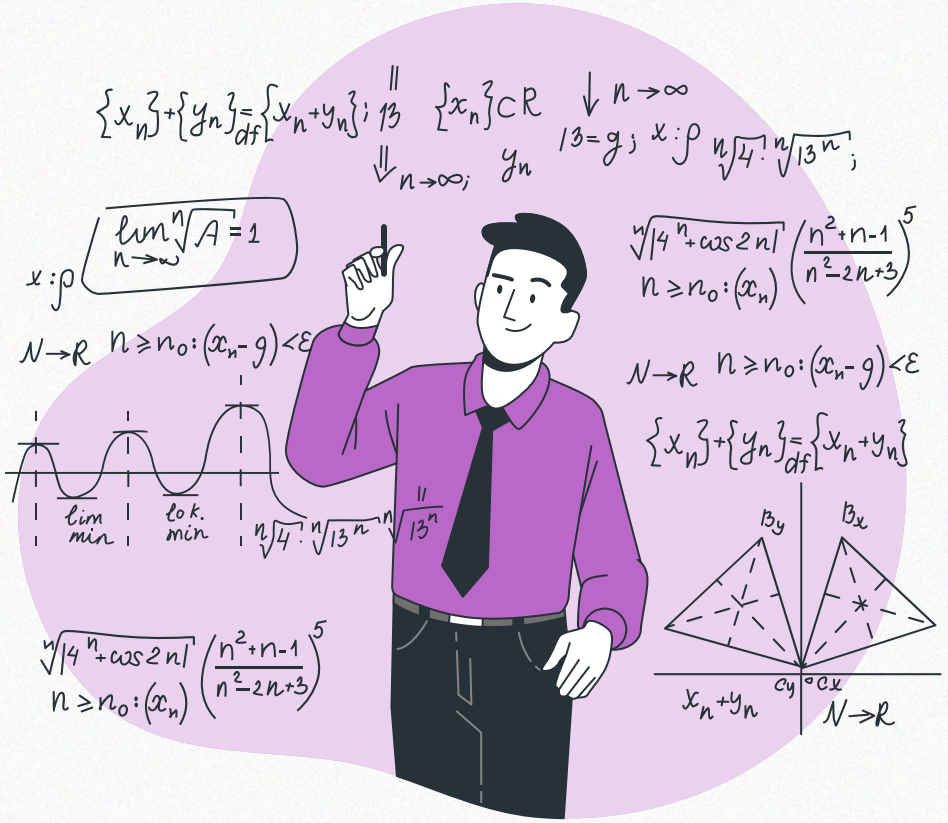
¿Los objetos dentro del agregado necesitan estar consistentes juntos?

Servicios de Dominio

Un **Servicio de Dominio** encapsula lógica que no pertenece naturalmente a una sola entidad o agregado.

Ejemplo
:

• Cálculo de multas por retrasos en devoluciones.



Cómo reconocerlas:

¿La lógica involucra múltiples entidades y no encaja bien en una sola?

Eventos de Dominio

Un **Evento de Dominio** representa un cambio significativo que ocurre dentro del dominio. Es una forma de comunicar que “algo importante ha sucedido”.

Ejemplo:
Cuando se presta un libro:

- **Evento:** `BookLoanCreatedEvent`
- **Datos relevantes:** ID del préstamo, ID del libro, ID del usuario, fecha límite.

Modelando

Entidades

- Estudiante
- Libro
- Prestamo

Value Objects

- Fecha de Devolucion
- Multa

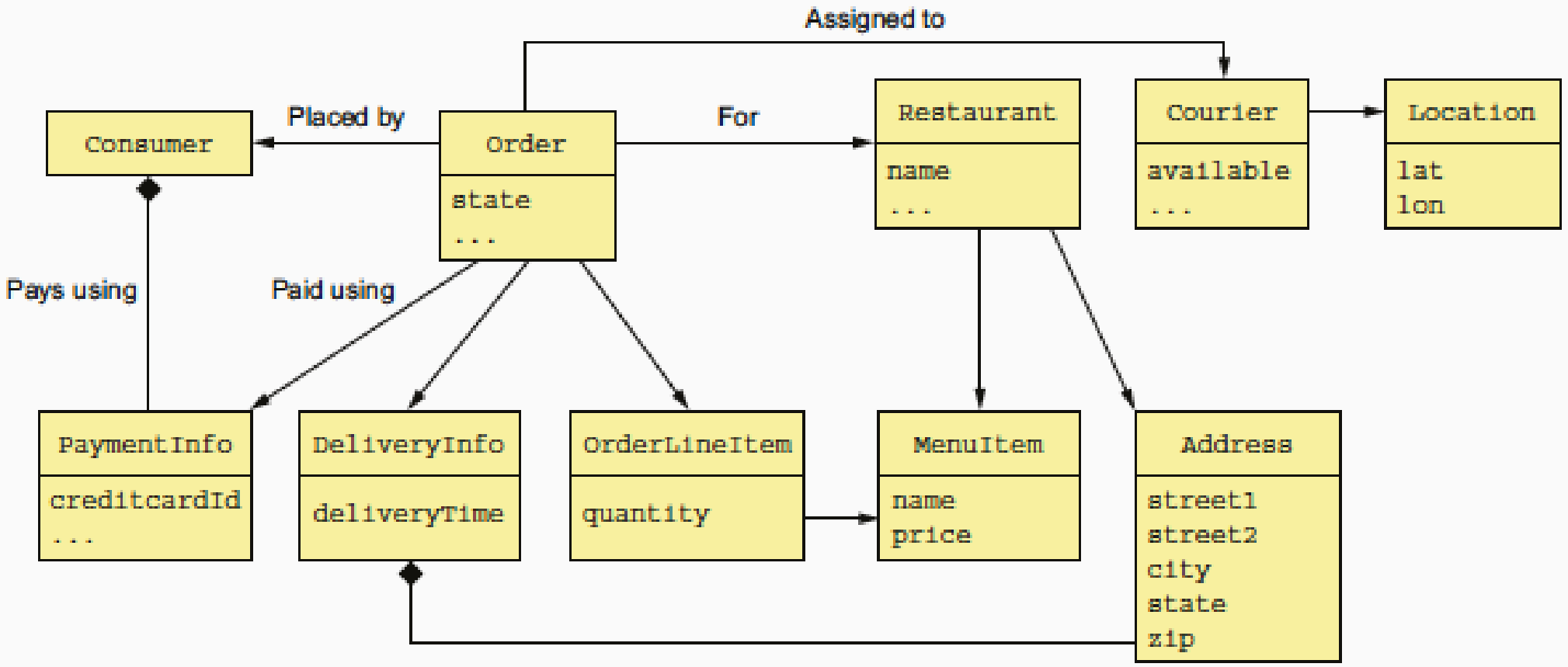
Servicios

- Calculador de Multas

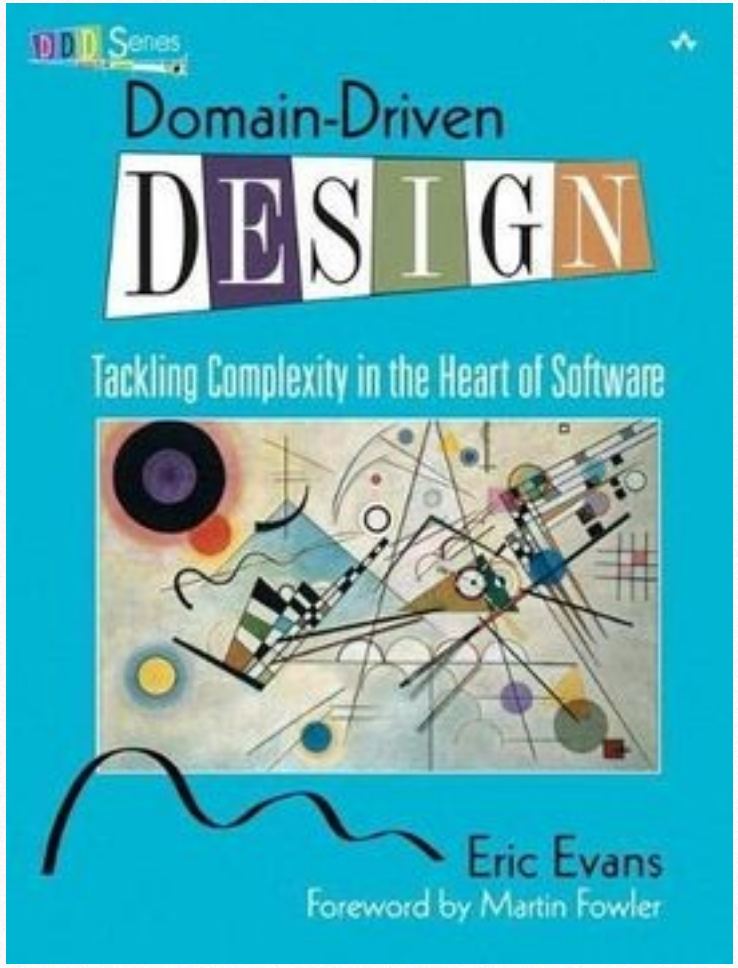
Eventos

- Prestamo
- Devolución

Modelando


Assigned to
Consumer
Placed by
Order
state
...
For
Restaurant
name
...
Courier
available
...
Location
lat
lon
Pays using
Paid using
PaymentInfo
creditcardId
...
DeliveryInfo
deliveryTime
OrderLineItem
quantity
MenuItem
name
price
Address
street1
street2
city
state
zip

Domain Driven Design


DDD Series
Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

¿Qué es Domain-Driven Design (DDD)?

Principios clave de DDD:

1. **Modelo rico en dominio:**
El modelo debe capturar las reglas, conceptos y relaciones clave del negocio.

2. **Ubiquitous Language:**
Un lenguaje común entre los desarrolladores y expertos del dominio.

3. **Contextos delimitados (Bounded Contexts):**
Áreas claramente definidas que encapsulan partes del dominio.

“Un enfoque de diseño de software que enfatiza la colaboración entre desarrolladores y expertos en el dominio para modelar un sistema basado en conceptos del negocio.”

Dominio: Biblioteca.

Términos del negocio:

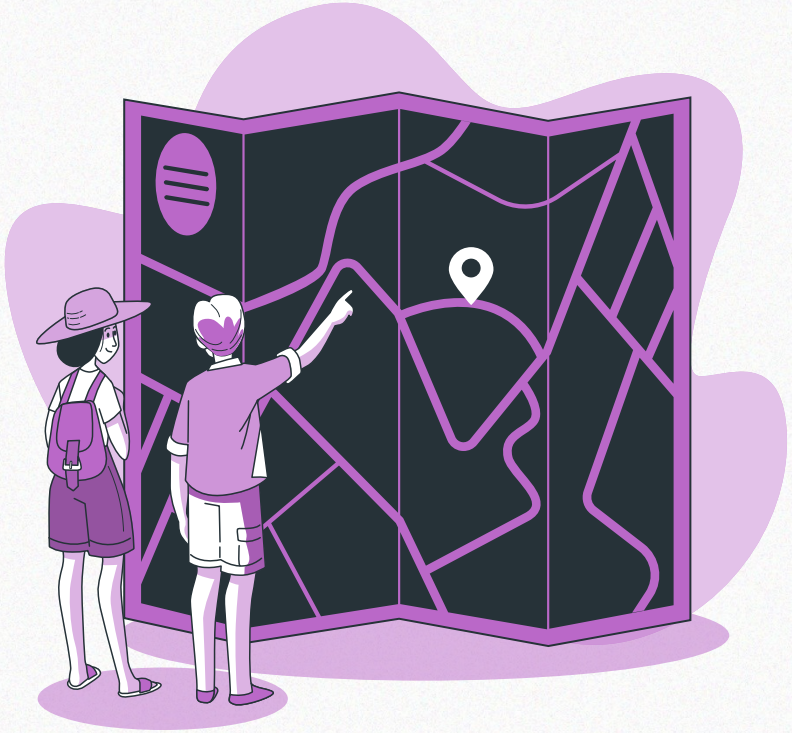
Préstamos, **Estudiantes**, **Multas**, **Libros**.

Lenguaje común: Un **préstamo** tiene una **fecha límite**, y si un **estudiante** no devuelve el **libro**, se genera una **multa**.

Dominio: Biblioteca.

¿Qué es un Bounded Context?

Un contexto delimitado encapsula un subconjunto del dominio con su propio lenguaje y reglas.

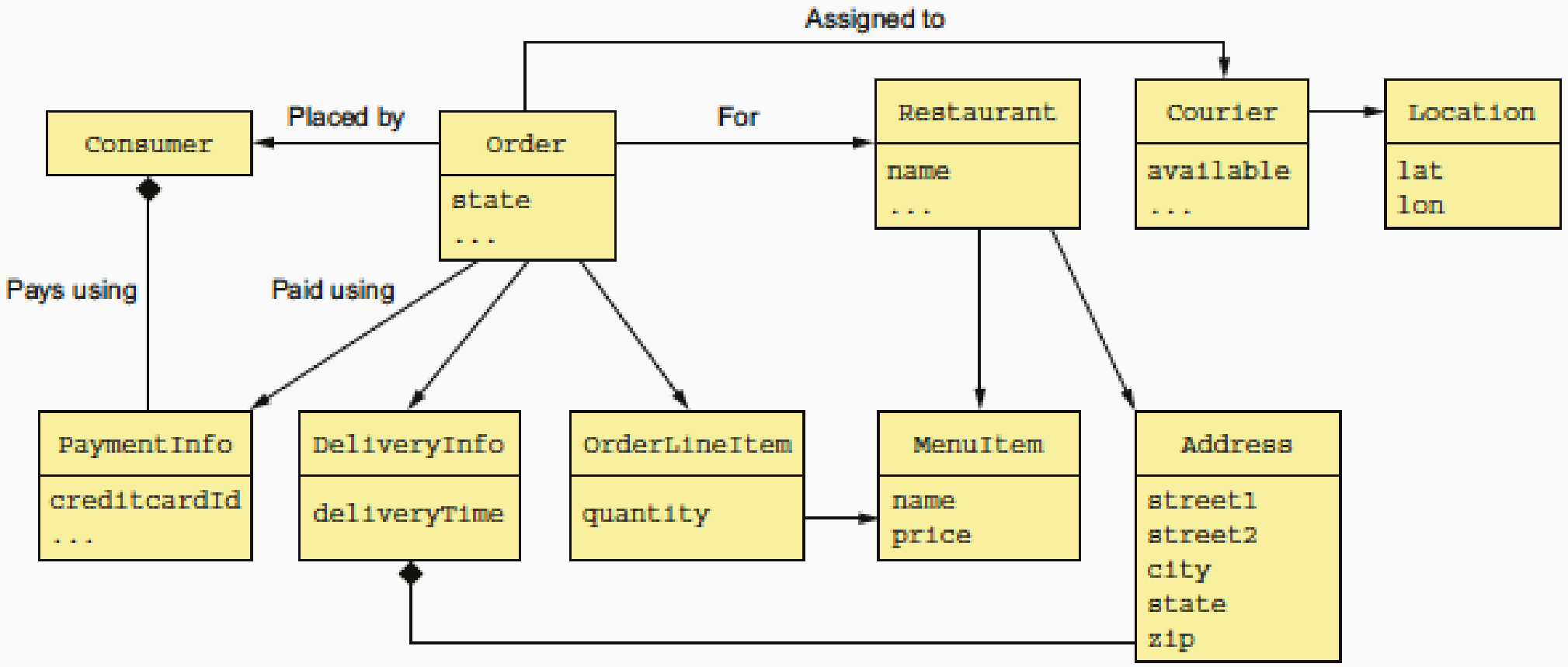


- Contextos delimitados:
- Gestión de Usuarios.
- Gestión de Inventario.
- Gestión de Préstamos.
- Gestión de Multas.

Trazado de Fronteras

- Clasificar relaciones como:
- **Fuertes:** Mantener en el mismo contexto.
- **Débiles:** Dividir en contextos separados.

Modelando


Assigned to
Consumer
Placed by
Order
state
...
For
Restaurant
name
...
Courier
available
...
Location
lat
lon
Pays using
Paid using
PaymentInfo
creditcardId
...
DeliveryInfo
deliveryTime
OrderLineItem
quantity
MenuItem
name
price
Address
street1
street2
city
state
zip

Modelando

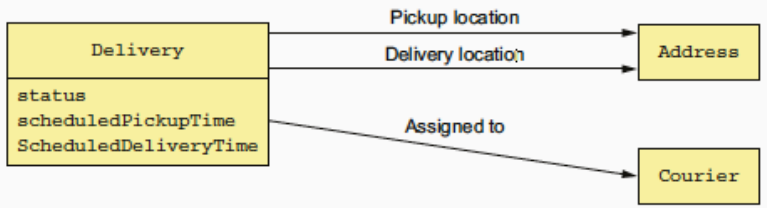


Figure 2.11 The Delivery Service domain model

Figure 2.12 The Kitchen Service domain model

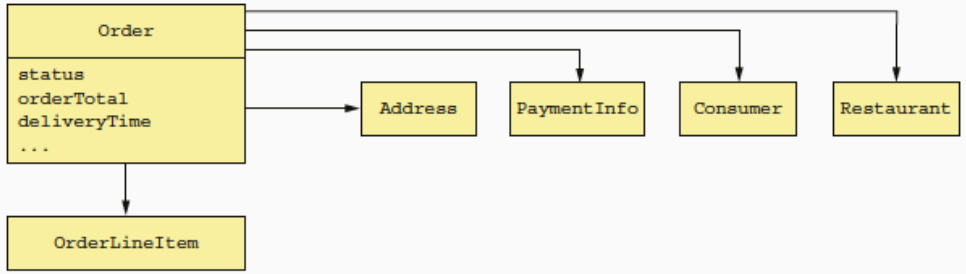


Figure 2.13 The Order Service domain model

Modelando

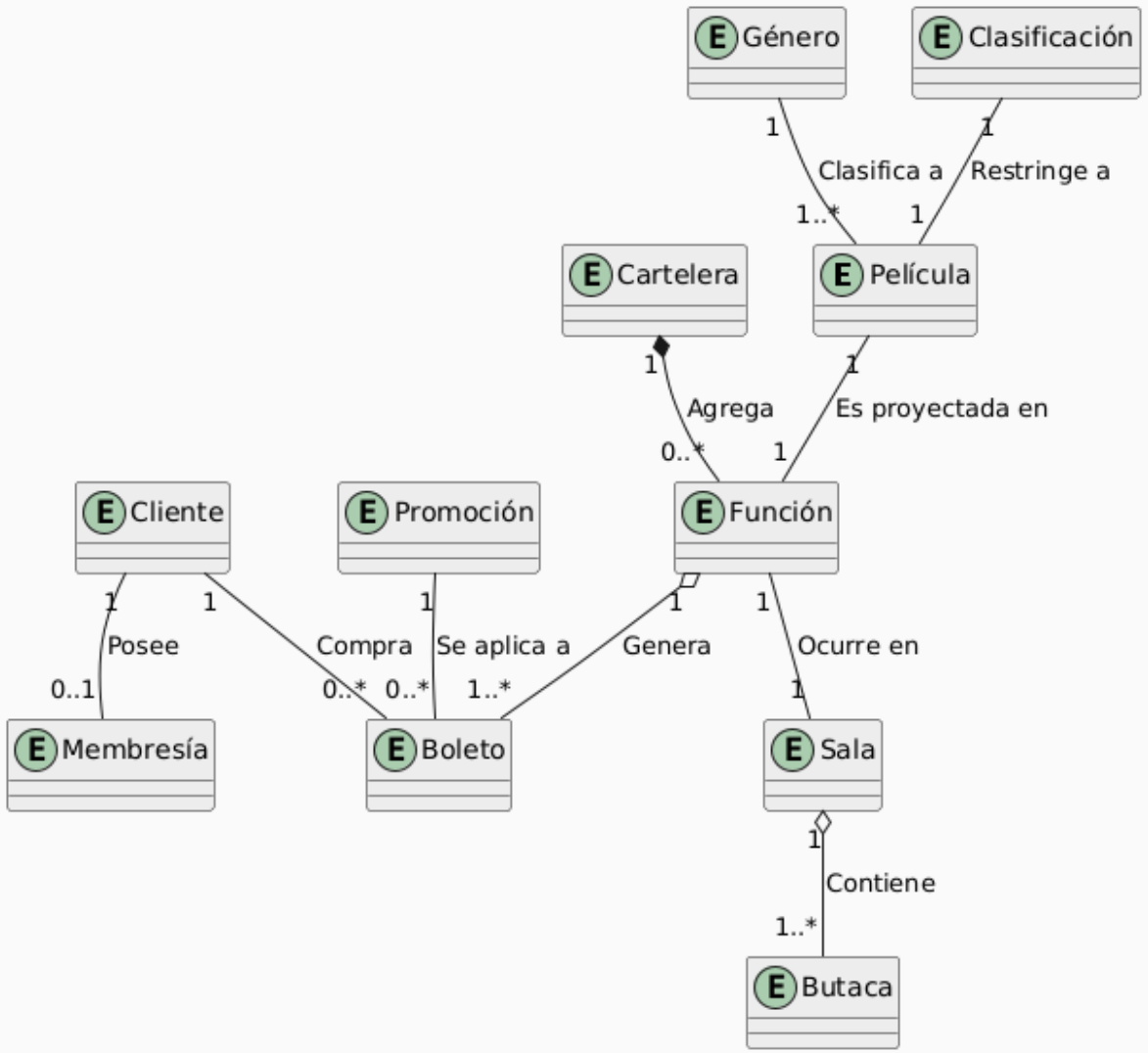
Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	▪ Consumer Service verifyConsumerDetails() ▪ Restaurant Service verifyOrderDetails() ▪ Kitchen Service createTicket() ▪ Accounting Service authorizeCard()
Restaurant Service	▪ findAvailableRestaurants() ▪ verifyOrderDetails()	—
Kitchen Service	▪ createTicket() ▪ acceptOrder() ▪ noteOrderReadyForPickup()	▪ Delivery Service scheduleDelivery()
Delivery Service	▪ scheduleDelivery() ▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()	—
Accounting Service	▪ authorizeCard()	—

Caso:
Modelar el Dominio de un Cine

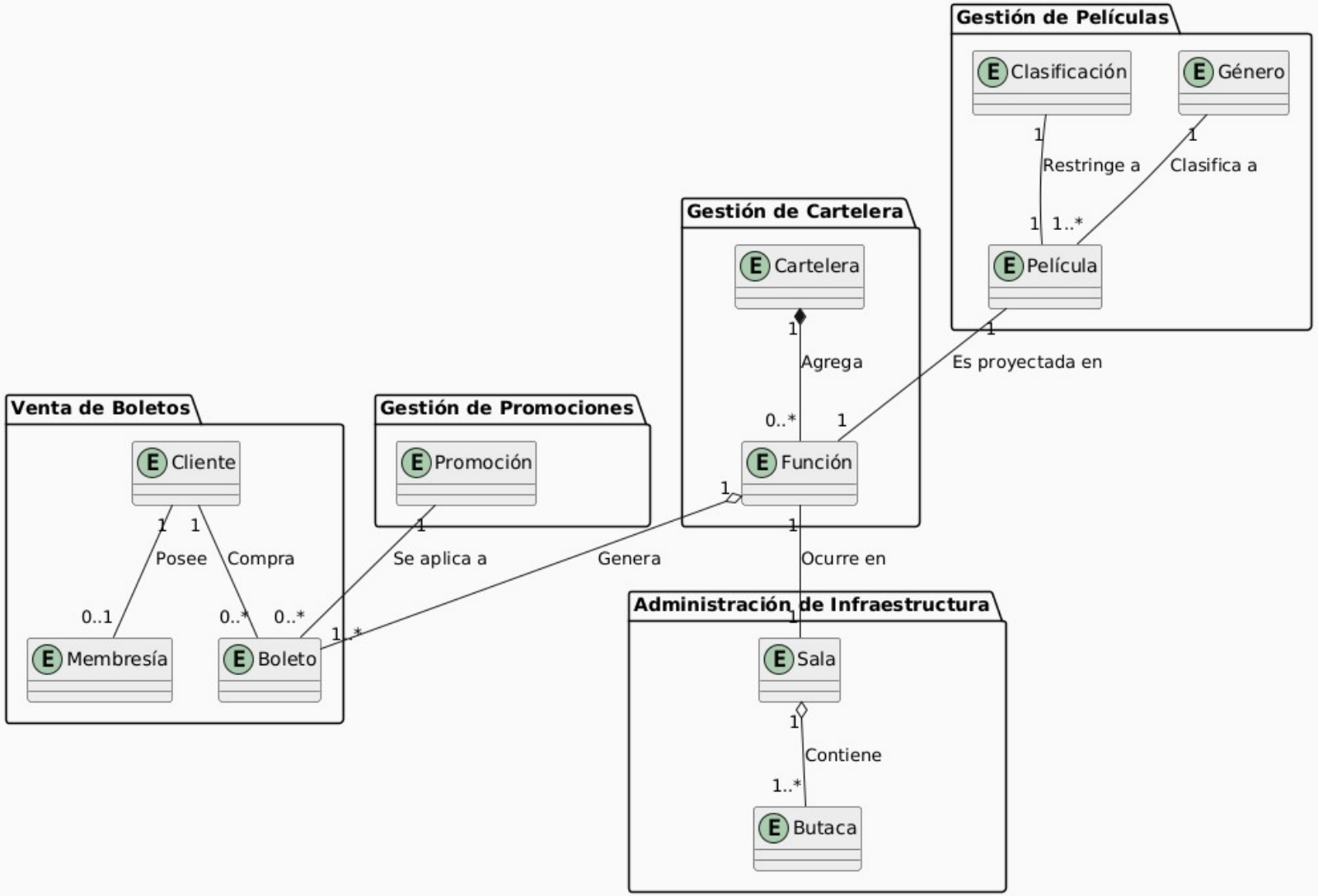
Elemento	Descripción
Película	Representa una película única con atributos como título, duración y director.
Sala	Espacio físico identificable con atributos como número, capacidad y ubicación.
Función	Una proyección específica de una película en una sala y horario.
Boleto	Entrada única vinculada a una función específica.
Cliente	Persona que interactúa con el sistema y realiza compras o reservas.
Promoción	Descuentos u ofertas aplicables a boletos o funciones bajo ciertas condiciones.

Elemento	Descripción
Género	Clasificación genérica de una película, como acción, comedia o drama.
Clasificación	Categoría de edad para una película, como PG, R, o G.
Membresía	Representa beneficios adicionales adquiridos por un cliente.
Butaca	Representa un asiento en una sala, asociado a un número y ubicación.
Cartelera	Representa la lista de funciones disponibles en un periodo específico.

Agregado	Raíz del Agregado	Descripción
Función	Función	Agrupa una película, sala, horario y las butacas disponibles.
Cliente	Cliente	Agrupa información del cliente, como sus compras y membresías.



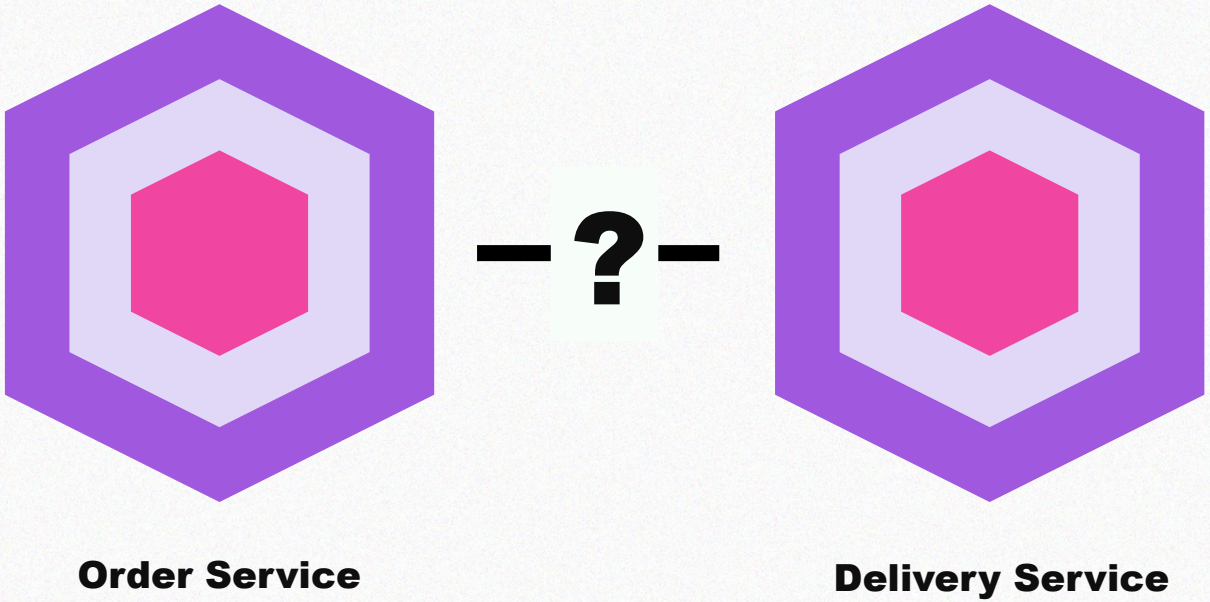

```
@startuml
entity "Película" as Pelicula
entity "Género" as Genero
entity "Clasificación" as Clasificacion
entity "Sala" as Sala
entity "Función" as Funcion
entity "Boleto" as Boleto
entity "Cliente" as Cliente
entity "Membresía" as Membresia
entity "Promoción" as Promocion
entity "Butaca" as Butaca
entity "Cartelera" as Cartelera

Pelicula "1" -- "1" Funcion : "Es proyectada en"
Genero "1" -- "1..*" Pelicula : "Clasifica a"
Clasificacion "1" -- "1" Pelicula : "Restringe a"
Sala "1" o-- "1..*" Butaca : "Contiene"
Funcion "1" -- "1" Sala : "Ocurre en"
Funcion "1" o-- "1..*" Boleto : "Genera"
Cliente "1" -- "0..*" Boleto : "Compra"
Cliente "1" -- "0..1" Membresia : "Posee"
Promocion "1" -- "0..*" Boleto : "Se aplica a"
Cartelera "1" *-- "0..*" Funcion : "Agrega"

@enduml
```


Gestión de Películas
Clasificación
Género
1
Restringe a
1
Clasifica a
1
1..*
Película
Gestión de Cartelera
Cartelera
1
Agrega
0..*
1
Función
1
Es proyectada en
Venta de Boletos
Cliente
1
1
Posee
Compra
0..1
0..*
0..*
Membresía
Boleto
1..*
Gestión de Promociones
Promoción
1
Se aplica a
1
Genera
1
Ocurre en
Administración de Infraestructura
1
Sala
1
Contiene
1..*
Butaca

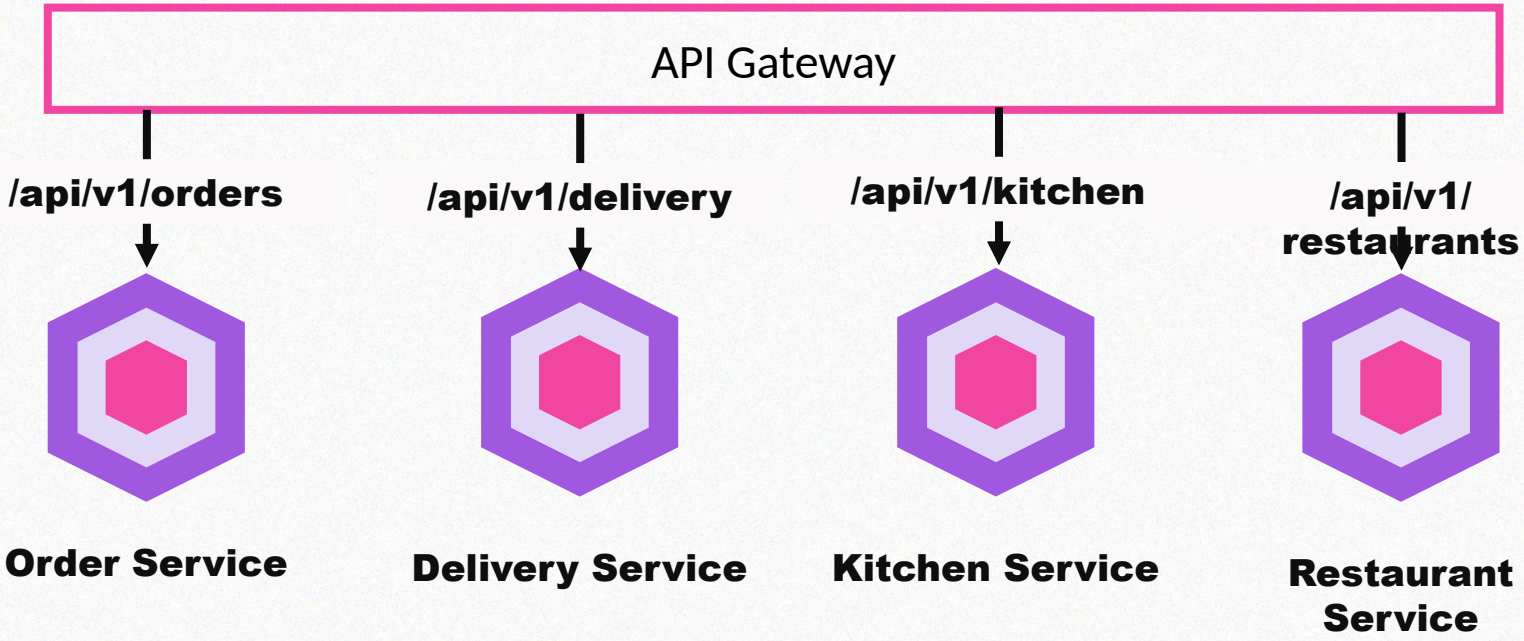
Comunicación de Microservicios

¿Como se comunica un microservicio con otro?

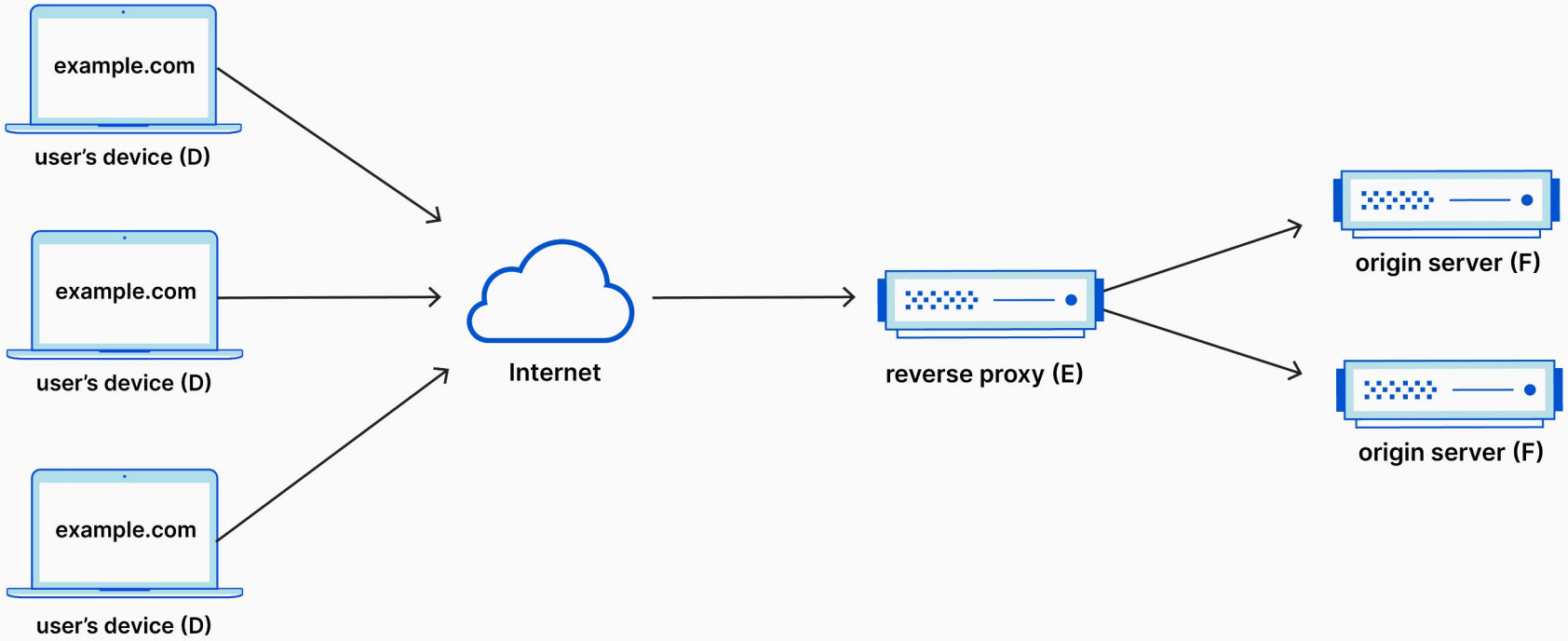
- **Sincrónica:** petición/respuesta (REST, gRPC, GraphQL).
- **Asíncrónica:** mensajes y eventos (Kafka, RabbitMQ, AMQP).



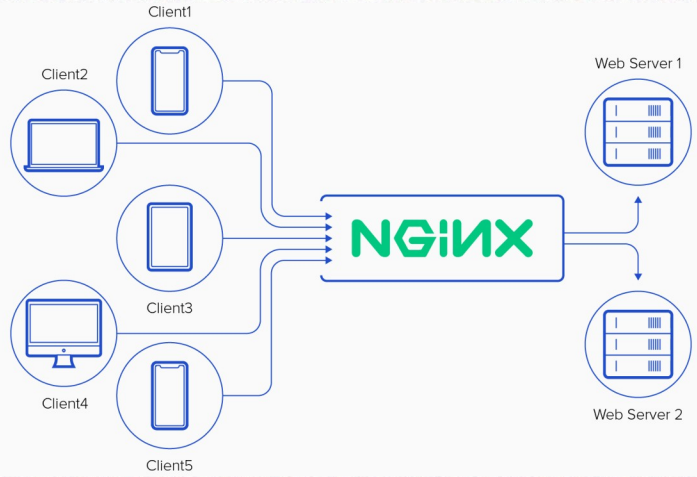
Comunicación sincronica (API)



Reverse Proxy Flow

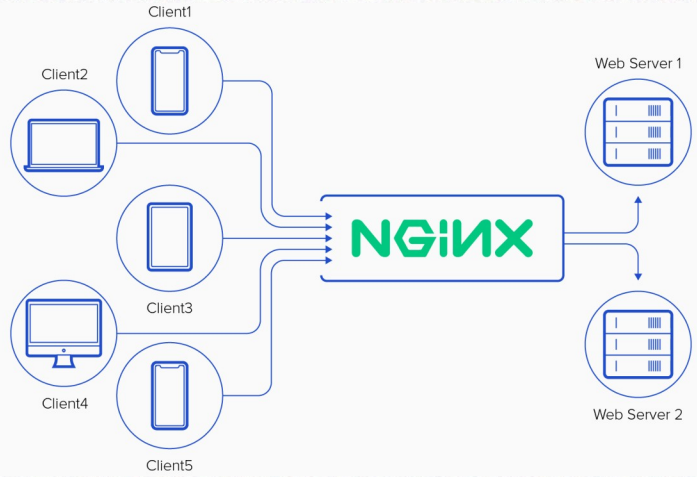


Comunicación sincronica (API)




```
server {
        listen 80;

        # Ruta para el servicio de pedidos
        location /orders {
            proxy_pass http://order_service;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        # Ruta para el servicio de restaurantes
        location /restaurants {
            proxy_pass http://restaurant_service;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        # Ruta para el servicio de entregas
        location /delivery {
            proxy_pass http://delivery_service;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        # Ruta para el servicio de cocina
        location /kitchen {
            proxy_pass http://kitchen_service;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
```

Comunicación sincronica (API)




```
upstream order_service {
    server localhost:8081;
    server localhost:8085;
}

upstream order_service {
    server localhost:8081;
    server localhost:8085;
}

upstream order_service {
    server localhost:8081 weight=3 fail_timeout=30s;
    server localhost:8085 weight=1;
}
```

Comunicación sincronica (API)

Protocolo HTTP

Metodos

GET

El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.

POST

El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.

PUT

El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.

DELETE

El método DELETE borra un recurso en específico.

PATCH

El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

Comunicación sincronica (API)

Protocolo HTTP

Headers

Accept-Language: en

Authorization: <auth-scheme> <authorization-parameters>

Content-Language: en-US

Comunicación sincronica (API)

Protocolo HTTP

Codigos de Error

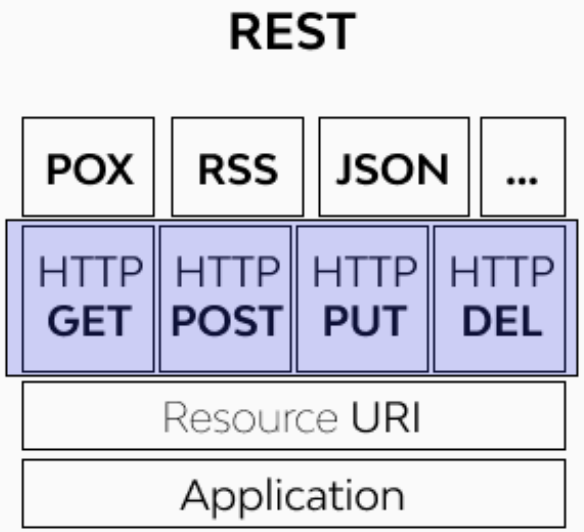
2xx Success	
200	Success / OK

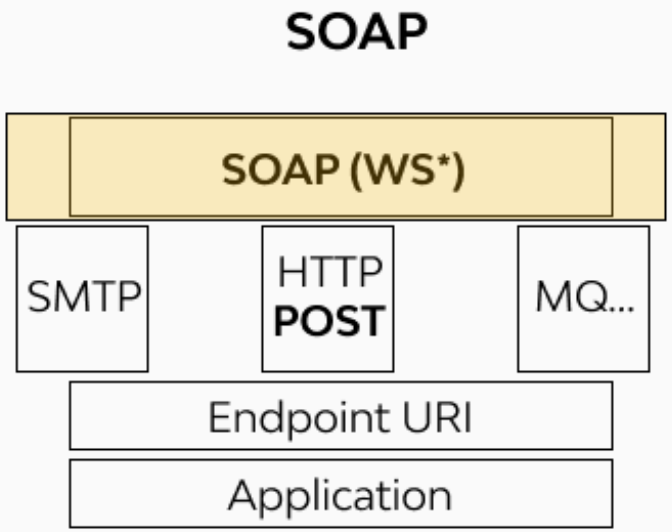
3xx Redirection	
301	Permanent Redirect
302	Temporary Redirect
304	Not Modified

4xx Client Error	
401	Unauthorized Error
403	Forbidden
404	Not Found
405	Method Not Allowed

5xx Server Error	
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

Comunicación sincronica (API)



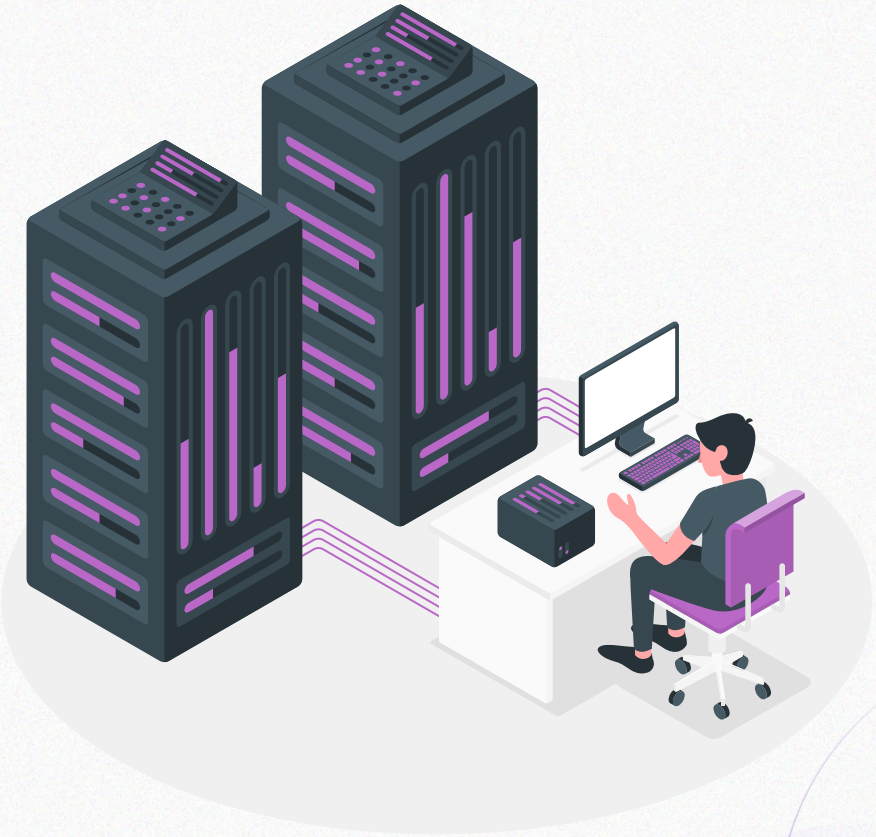


Comunicación sincronica (API)

REST (Representational State Transfer)

Principios:

- **Recursos:** Cualquier cosa que se pueda identificar (usuarios, productos, artículos).
- **Colecciones:** Conjuntos de recursos
- **URIs:** Identificadores únicos para los recursos.
- **Métodos HTTP:** Acciones sobre los recursos (GET para obtener, POST para crear, etc.).
- **Representaciones:** Formatos de intercambio de información (JSON, XML).
- **Sin estado:** Cada solicitud contiene toda la información necesaria.



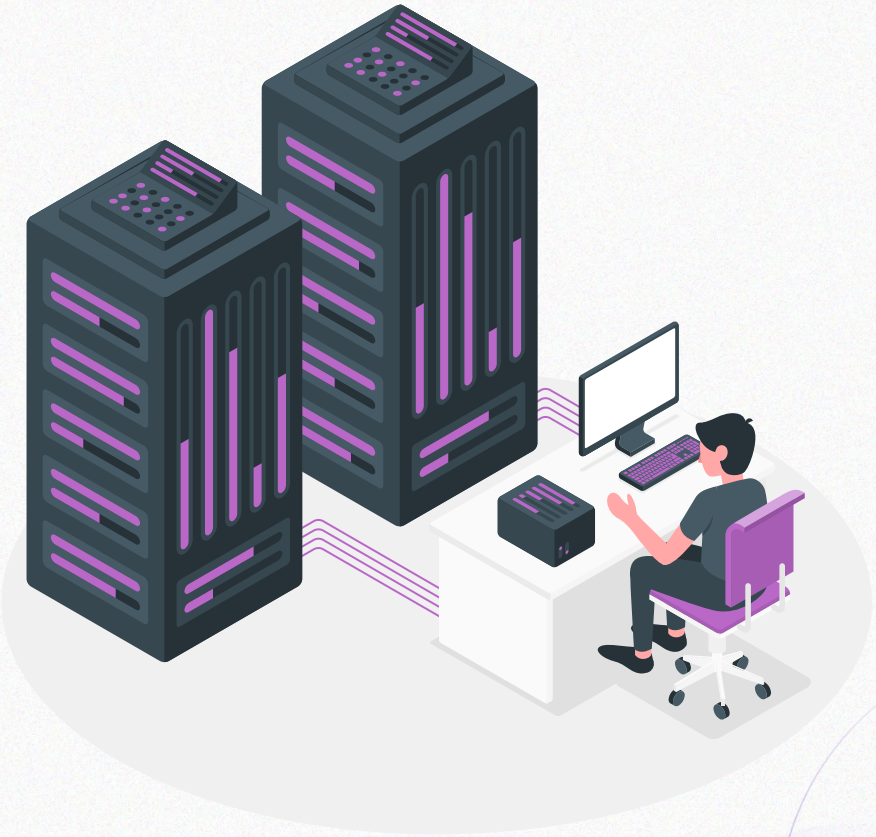
Comunicación sincronica (API)

SOAP (Simple Object Access Protocol)

HTTP POST: Aunque SOAP puede usar otros protocolos, POST es el más común. Esto significa que los mensajes SOAP generalmente se envían como solicitudes POST a un servidor web.

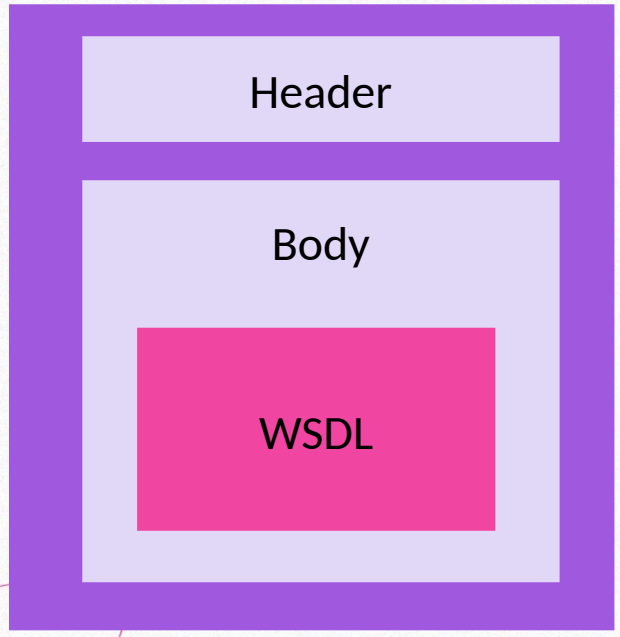
Endpoint URI: Similar a REST, SOAP utiliza URIs para identificar dónde se encuentra el servicio web al que se envía la solicitud.

Formato: SOAP utiliza XML para estructurar los mensajes. Esto define un formato muy específico para las solicitudes y respuestas.



Comunicación sincronica (API)

SOAP (Simple Object Access Protocol)




```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   <soapenv:Header>
      <tem:AuthHeader>
         <tem:Username>usuario</tem:Username>
         <tem:Password>contraseña</tem:Password>
      </tem:AuthHeader>
   </soapenv:Header>
   <soapenv:Body>
      <tem:CrearUsuario>
         <tem:Usuario>
            <tem:Nombre>Juan Perez</tem:Nombre>
            <tem:Edad>30</tem:Edad>
            <tem:Email>juan.perez@example.com</tem:Email>
            <tem:Direcciones>
               <tem:Direccion>
                  <tem:Calle>Calle Principal 123</tem:Calle>
                  <tem:Ciudad>Ciudad A</tem:Ciudad>
               </tem:Direccion>
               <tem:Direccion>
                  <tem:Calle>Calle Secundaria 456</tem:Calle>
                  <tem:Ciudad>Ciudad B</tem:Ciudad>
               </tem:Direccion>
            </tem:Direcciones>
         </tem:Usuario>
      </tem:CrearUsuario>
   </soapenv:Body>
</soapenv:Envelope>
```

Comunicación sincronica (API)

gRPC (Remote Procedure Calls)

Es un framework de código abierto de alto rendimiento que permite a las aplicaciones comunicarse entre sí de manera eficiente y en tiempo real. A diferencia de REST y SOAP, que se basan en mensajes, gRPC se centra en llamadas a procedimientos remotos (RPC). Esto significa que una aplicación puede llamar a una función en otra aplicación como si fuera una función local.

Comunicación sincronica (API)

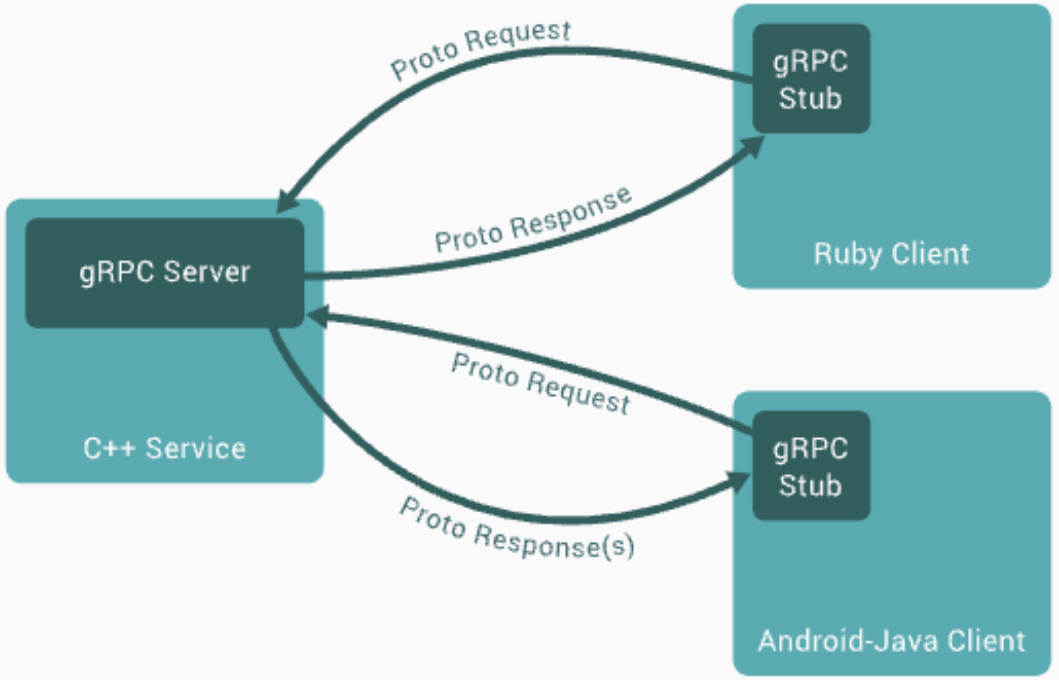
gRPC (Remote Procedure Calls)

- **Protocol Buffers:** gRPC utiliza Protocol Buffers (protobuf) para serializar los datos. Protobuf es un mecanismo eficiente y compacto para serializar datos estructurados, más eficiente que XML.
- **HTTP/2:** gRPC se basa en HTTP/2, que ofrece características como multiplexación de streams, compresión de cabeceras y comunicación bidireccional. Esto lo hace más rápido y eficiente que HTTP/1.1, que se usa en REST y SOAP.
- **IDL:** gRPC utiliza un lenguaje de definición de interfaces (IDL) para definir los servicios y los mensajes que se intercambian. Este IDL se usa para generar código en diferentes lenguajes de programación.

Comunicación sincronica (API)

Funcionamiento

- **Definición de servicios:** Se define un archivo .proto que describe los servicios y los mensajes utilizando el IDL de gRPC.
- **Generación de código:** Se utiliza la herramienta protoc para generar código en diferentes lenguajes de programación (como Java, Python, Go) a partir del archivo .proto.
- **Llamadas a procedimientos remotos:** Las aplicaciones cliente y servidor utilizan el código generado para comunicarse entre sí. El cliente realiza una llamada a un procedimiento remoto en el servidor, y el servidor responde con un resultado.



Comunicación sincronica (API)

hola.proto

```
syntax = "proto3";
package hola;

service HolaService {
        rpc Saludar (HolaRequest) returns (HolaReply) {} }

message HolaRequest { string nombre = 1; }

message HolaReply { string saludo = 1; }
```

Comunicación sincronica (API)

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. hola.proto
```

`hola_pb2.py`	Las definiciones de mensajes
`hola_pb2_grpc.py`	Las definiciones de servicios

Comunicación sincronica (API)

server.py

```
import grpc from concurrent

import futures import hola_pb2
import hola_pb2_grpc

class HolaService(hola_pb2_grpc.HolaServiceServicer):

    def Saludar(self, request, context):
            return hola_pb2.HolaReply(saludo='Hola, %s!' % request.nombre)

    def serve():
            server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    hola_pb2_grpc.add_HolaServiceServicer_to_server(HolaService(), server)

        server.add_insecure_port('[::]:50051’)
        server.start()
        server.wait_for_termination()
```

Comunicación sincronica (API)

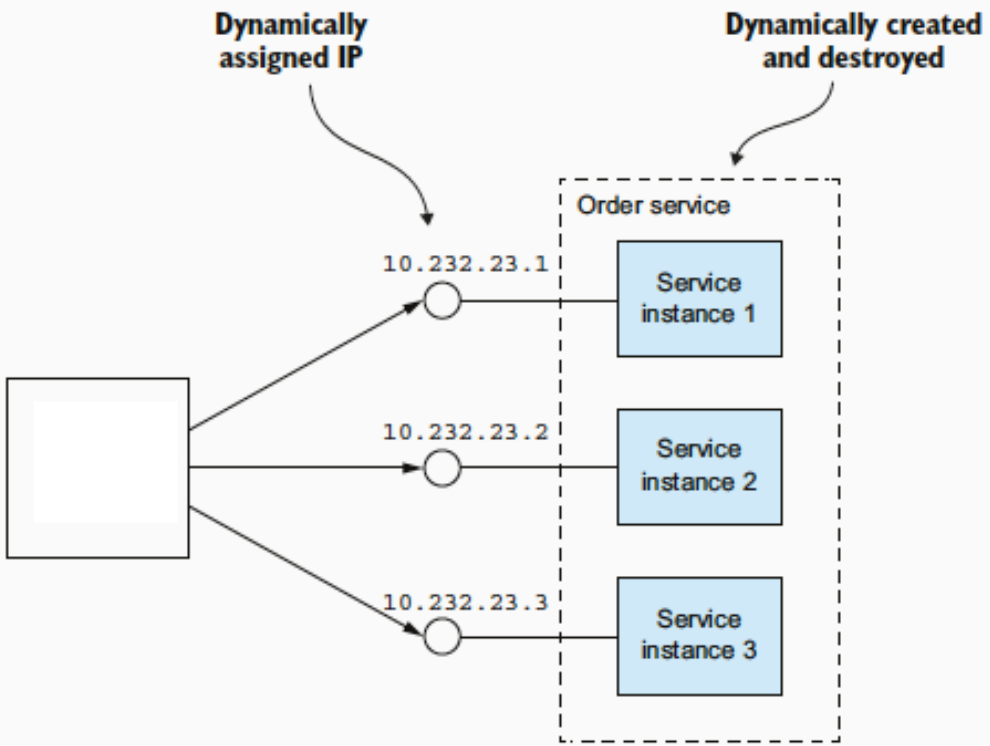
client.py

```
import grpc import hola_pb2
import hola_pb2_grpc

def run():
        with grpc.insecure_channel('localhost:50051') as channel:

        stub = hola_pb2_grpc.HolaServiceStub(channel)
        response = stub.Saludar(hola_pb2.HolaRequest(nombre='Mundo’))

        print("Respuesta del servidor: " + response.saludo)
```

Descubrimiento de Servicios

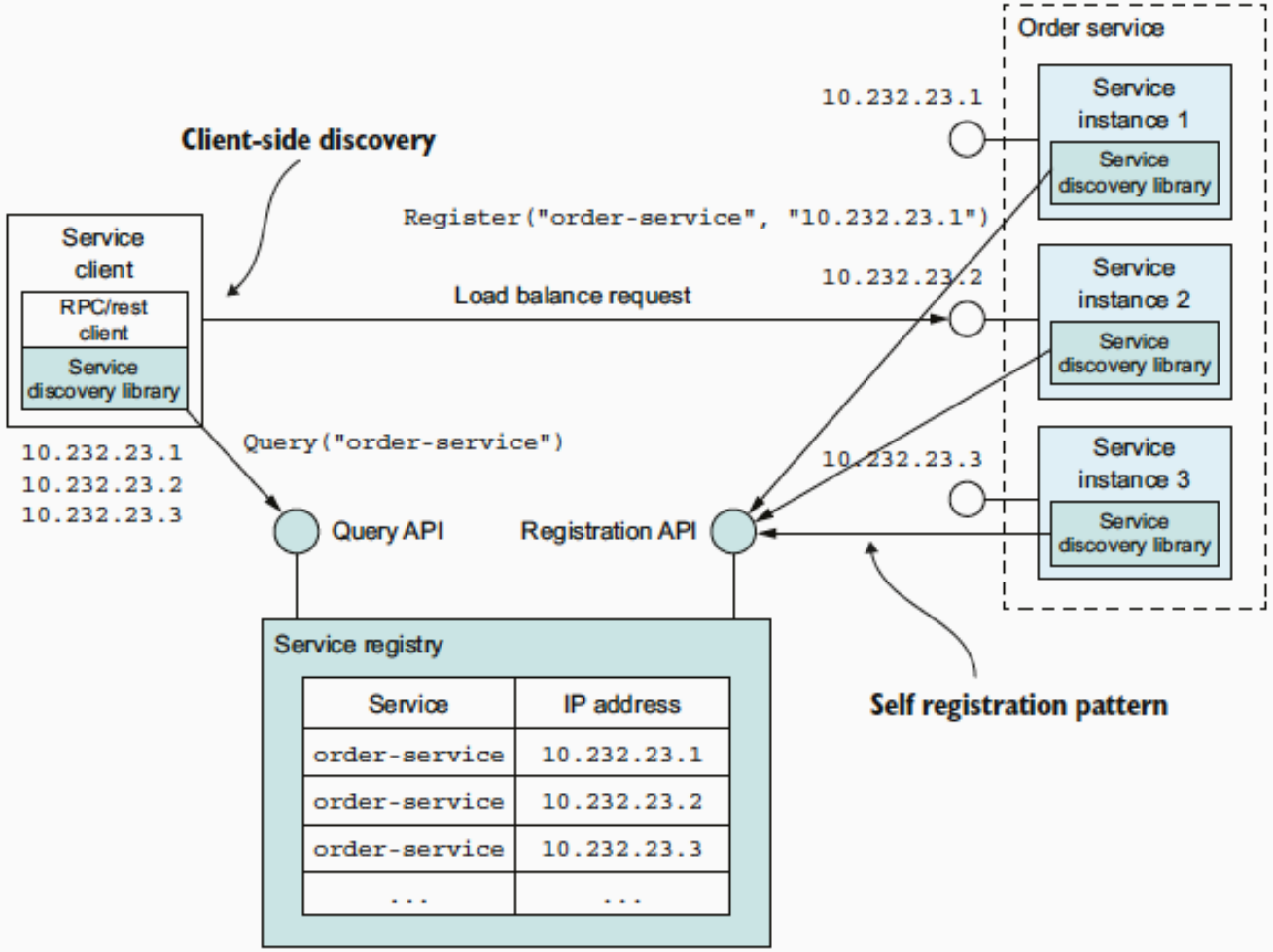
Si los servicios se despliegan dinamicamente,

¿Cómo se a donde hacer la peticion?

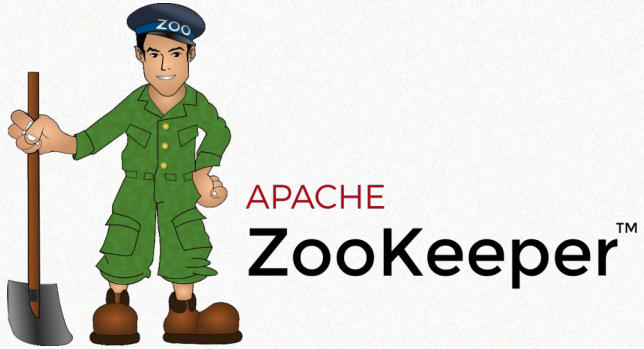


Descubrimiento de Servicios

El **descubrimiento de servicios** permite que un microservicio localice la dirección (URL o IP) de otro microservicio en una arquitectura distribuida. Esto es especialmente importante en un entorno donde los servicios son dinámicos, pueden cambiar de ubicación (IP) y se escalan de forma independiente.



Descubrimiento de Servicios



Descubrimiento de Servicios

```python
import requests
import time

SERVICE_REGISTRY_URL = "http://localhost:8761/eureka/apps"  # URL del registro de servicios
SERVICE_NAME = "ORDER-SERVICE"
SERVICE_INSTANCE_ID = "order-service-1"
SERVICE_IP = "10.232.23.1"
SERVICE_PORT = 8080

def register_service():
    data = {
        "instance": {
            "instanceId": SERVICE_INSTANCE_ID,
            "hostName": SERVICE_IP,
            "app": SERVICE_NAME,
            "ipAddr": SERVICE_IP,
            "port": {"$": SERVICE_PORT, "@enabled": "true"},
            "status": "UP",
        }
    }
    headers = {"Content-Type": "application/json"}
    response = requests.post(f"{SERVICE_REGISTRY_URL}/{SERVICE_NAME}", json=data, headers=headers)
    if response.status_code == 204:
        print(f"Service {SERVICE_NAME} registered successfully.")
    else:
        print(f"Failed to register service: {response.status_code}")
```

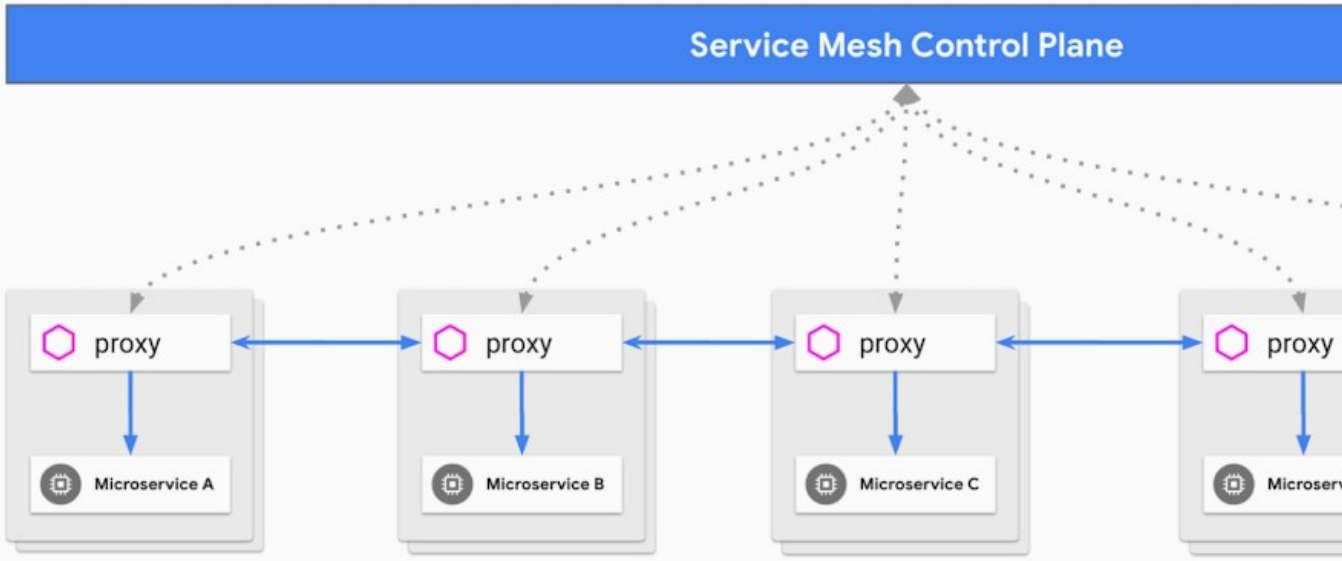
Descubrimiento de Servicios

```python
import requests

SERVICE_REGISTRY_URL = "http://localhost:8761/eureka/apps"  # Registro de servicios
SERVICE_NAME = "ORDER-SERVICE"

def get_service_instance(service_name):
    response = requests.get(f"{SERVICE_REGISTRY_URL}/{service_name}")
    if response.status_code == 200:
        data = response.json()
        instances = data["application"]["instance"]
        return [f"{instance['ipAddr']}:{instance['port']['$']}" for instance in instances]
    else:
        print(f"Failed to fetch service instances: {response.status_code}")
        return []

# Obtener una instancia y hacer la solicitud
instances = get_service_instance(SERVICE_NAME)
if instances:
    service_url = f"http://{instances[0]}/orders"
    print(f"Requesting {service_url}")
    response = requests.get(service_url)
    print(response.json())
else:
    print("No instances available.")
```

Service Mesh

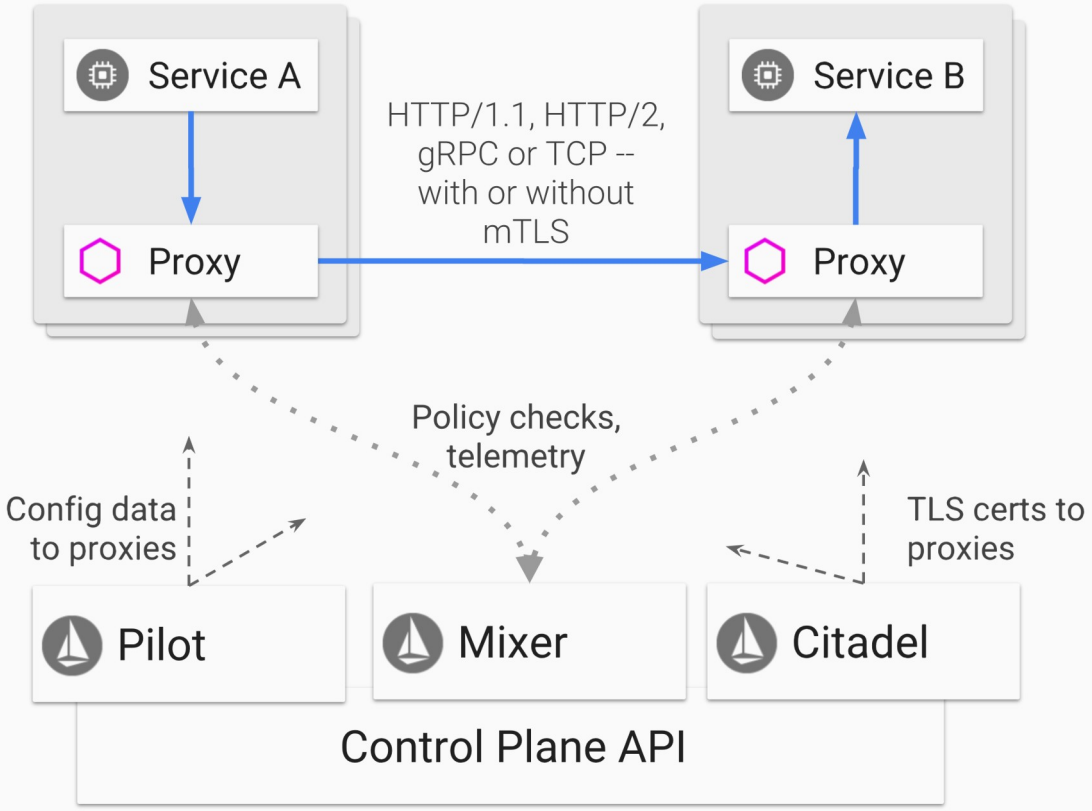
Un **service mesh** es una capa de red lógica que permite gestionar y optimizar la comunicación entre microservicios. Dentro de esta infraestructura, los **meshables** son componentes o patrones que permiten que los microservicios participen en el service mesh, habilitando funcionalidades como:

1. **Descubrimiento automático de servicios**.
2. **Balanceo de carga avanzado**.
3. **Encriptación de tráfico (TLS mutuo)**.
4. **Retrys, circuit breakers y timeouts**.
5. **Observabilidad (tracing, métricas y logs)**.



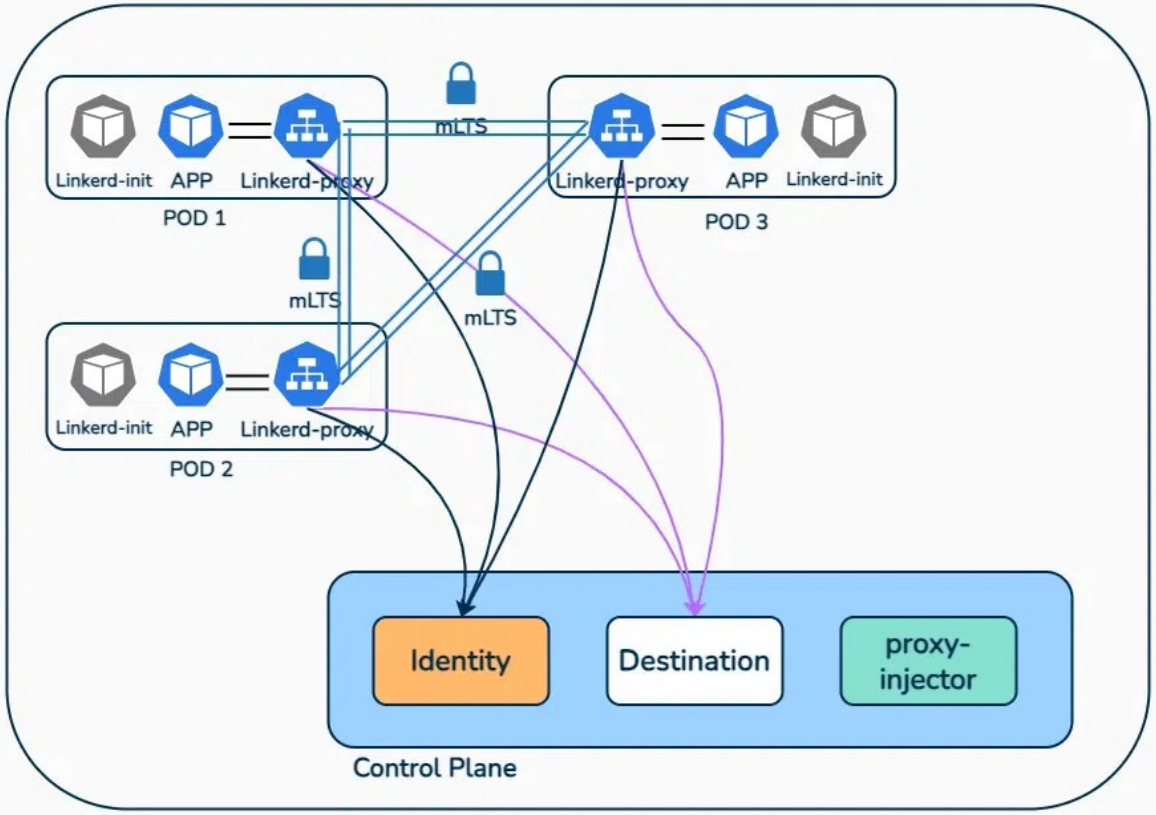
Meshables: Comunicación y Descubrimiento en un Service Mesh

En un service mesh, los meshables permiten una comunicación más eficiente al abstraer la lógica de red y manejarla en un plano separado (data plane y control plane). Este enfoque es muy útil en entornos dinámicos con alta complejidad.



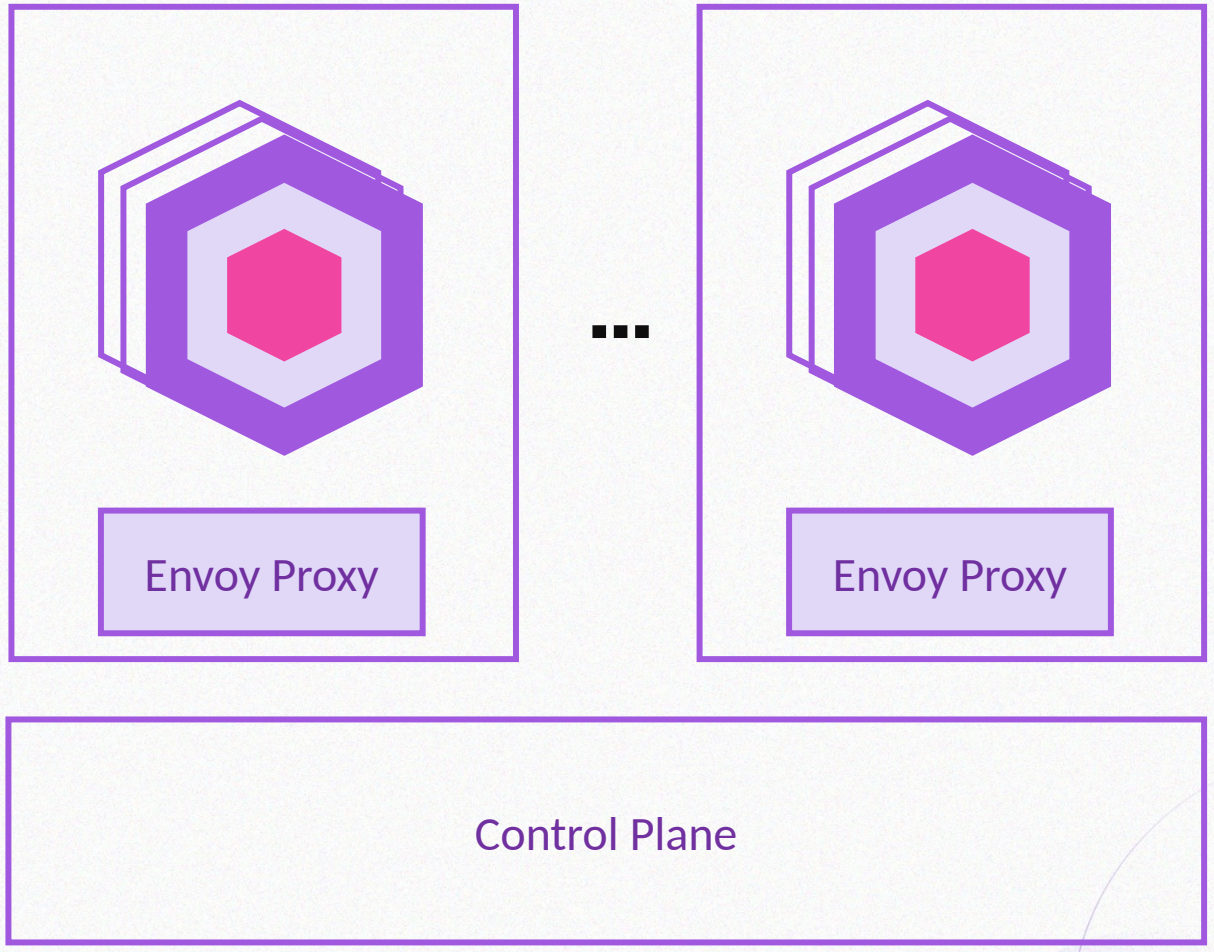
Meshables: Comunicación y Descubrimiento en un Service Mesh

En un service mesh, los meshables permiten una comunicación más eficiente al abstraer la lógica de red y manejarla en un plano separado (data plane y control plane). Este enfoque es muy útil en entornos dinámicos con alta complejidad.



Meshables: Comunicación y Descubrimiento en un Service Mesh

En un service mesh, los meshables permiten una comunicación más eficiente al abstraer la lógica de red y manejarla en un plano separado (data plane y control plane). Este enfoque es muy útil en entornos dinámicos con alta complejidad.



Meshables: Comunicación y Descubrimiento en un Service Mesh

```
String userResponse = webClient
        .get()
        .uri("http://user-service.default.svc.cluster.local/user/" + userId)
        .retrieve()
        .bodyToMono(String.class)
        .block(); // Bloquea hasta obtener respuesta
```

CODIGO
powered by Tecsup/>

LINKERD

Istio

kiali

anonymous

Overview

Graph

Applications

Workloads

Services

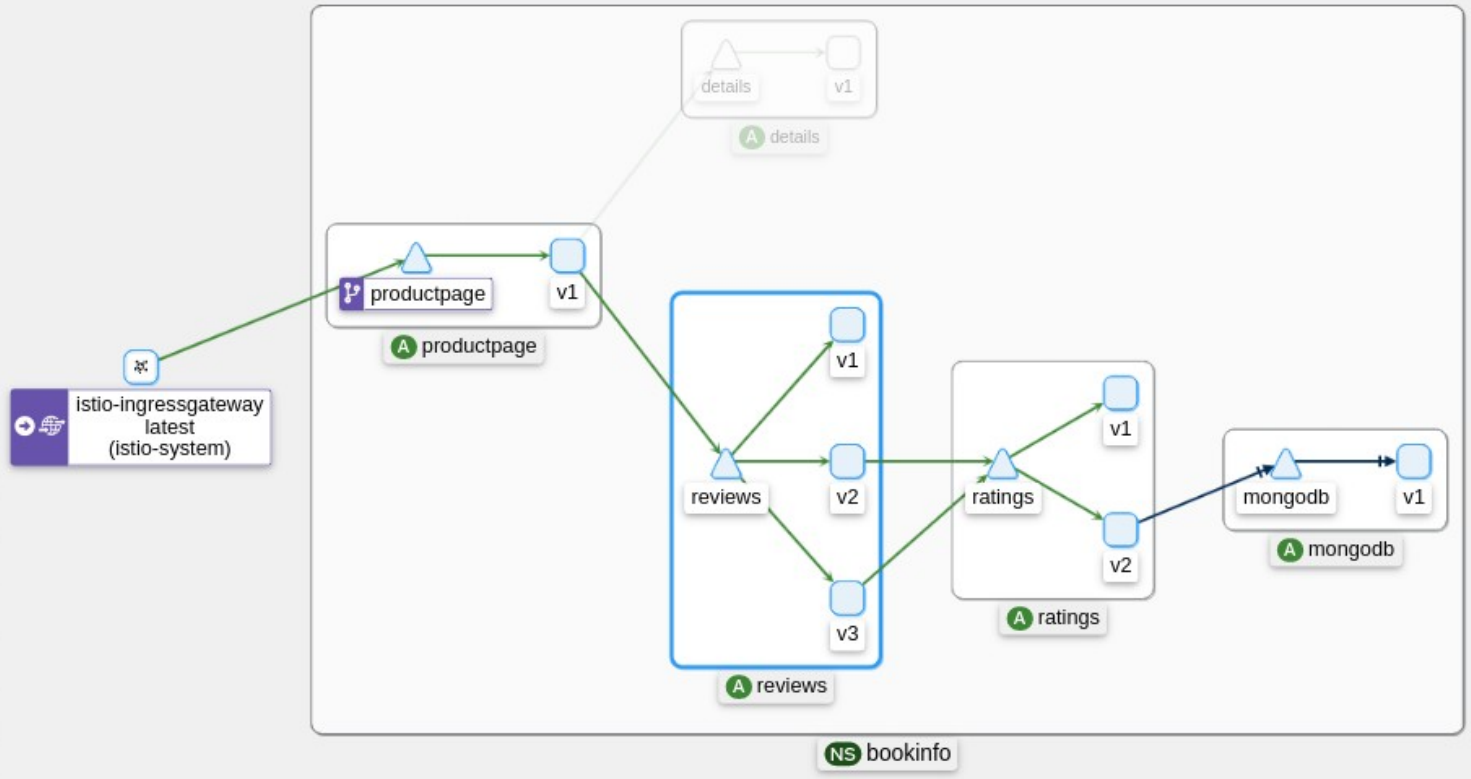
Istio Config

Namespace: bookinfo | Traffic | Versioned app graph

Last 1m | Every 1m

Display | Find... | Hide...

Nov 15, 02:06:02 PM ... 02:07:02 PM



Application

NS bookinfo

A reviews

health

S reviews

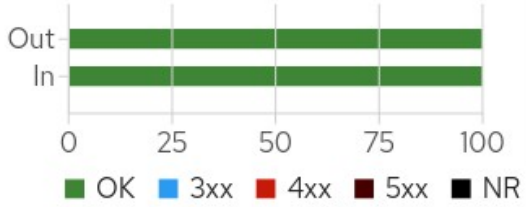
W reviews-v1

W reviews-v2

W reviews-v3

HTTP (requests per second):

	Total	%Success	%Error
In	0.89	100.00	0.00
Out	0.55	100.00	0.00



HTTP - Inbound Request Traffic min / max:

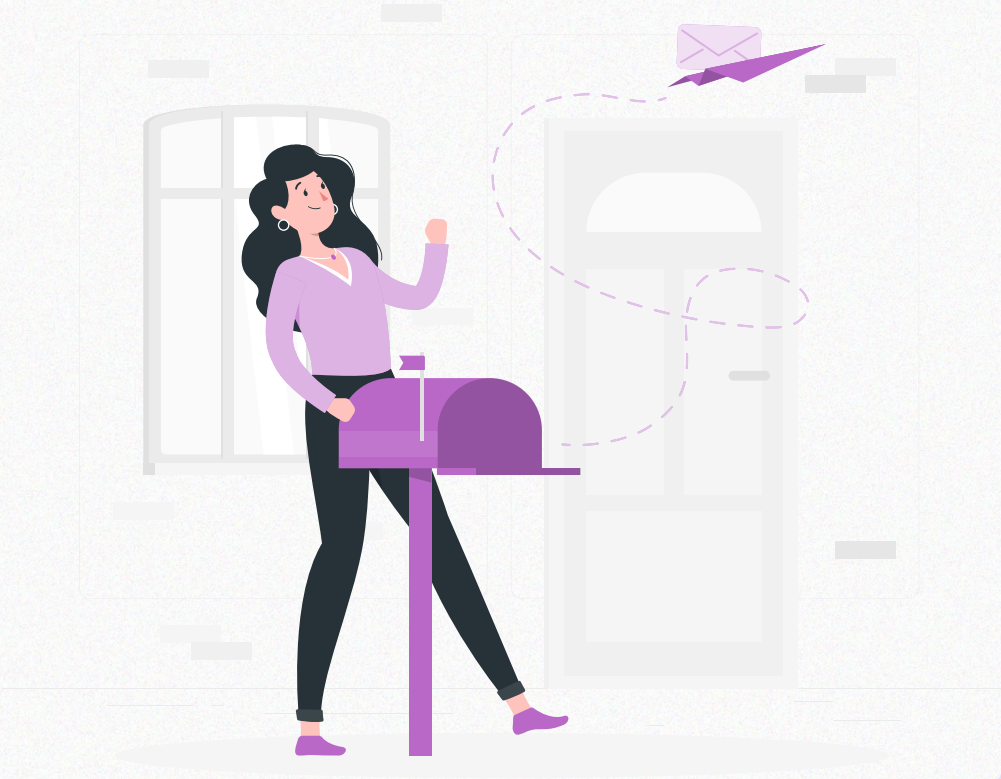
RPS: 0.87 / 1.13 , %Error 0.00 / 0.00

Hide

Comunicación Asincrona

La comunicación asíncrona permite que los servicios se comuniquen sin esperar una respuesta inmediata.

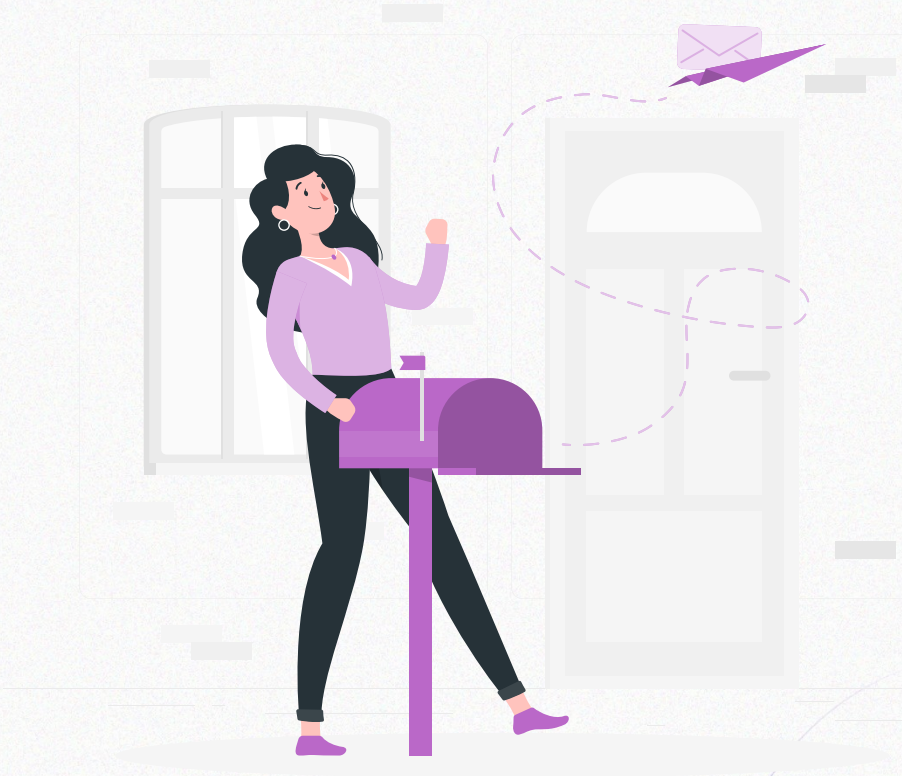
En lugar de bloquearse, los servicios envían mensajes a un intermediario (como un *message broker*), que se encarga de distribuirlos a los servicios correspondientes.



Comunicación Asincrona

Componentes Clave

1. **Message Broker:** Actúa como intermediario para transmitir mensajes entre servicios.
    - Ejemplos: **RabbitMQ**, **Kafka**, **Amazon SQS**, **Google Pub/Sub**.
2. **Colas de Mensajes:** Los mensajes enviados por un servicio productor se almacenan temporalmente en una cola hasta que un consumidor los procese.
3. **Eventos:** Los servicios publican eventos en un tema (*topic*), y otros servicios interesados los consumen.
    - Ejemplo: Cuando un pedido es confirmado, el servicio de pedidos puede emitir un evento “PedidoConfirmado”, que otros servicios (facturación, inventario) escuchan.
4. **Modelo de Comunicación**
    - **Pub/Sub (Publicación/Suscripción):** Varios consumidores pueden suscribirse a un tema para recibir eventos.
    - **Point-to-Point:** Un mensaje en una cola es consumido por un solo servicio.



Comunicación Asincronica (API)

La comunicación asíncrona en arquitecturas de microservicios es un enfoque que permite a los servicios interactuar sin necesidad de esperar respuestas inmediatas, lo que mejora la escalabilidad y la resiliencia del sistema.

Ventajas de la Comunicación Asíncrona

- **Desacoplamiento:** Los microservicios pueden operar independientemente, lo que reduce la dependencia entre ellos.
- **Escalabilidad:** Permite manejar un mayor volumen de solicitudes sin bloquear el procesamiento.
- **Tolerancia a fallos:** Un microservicio puede seguir funcionando aunque otros estén inactivos

Estilos de Comunicación

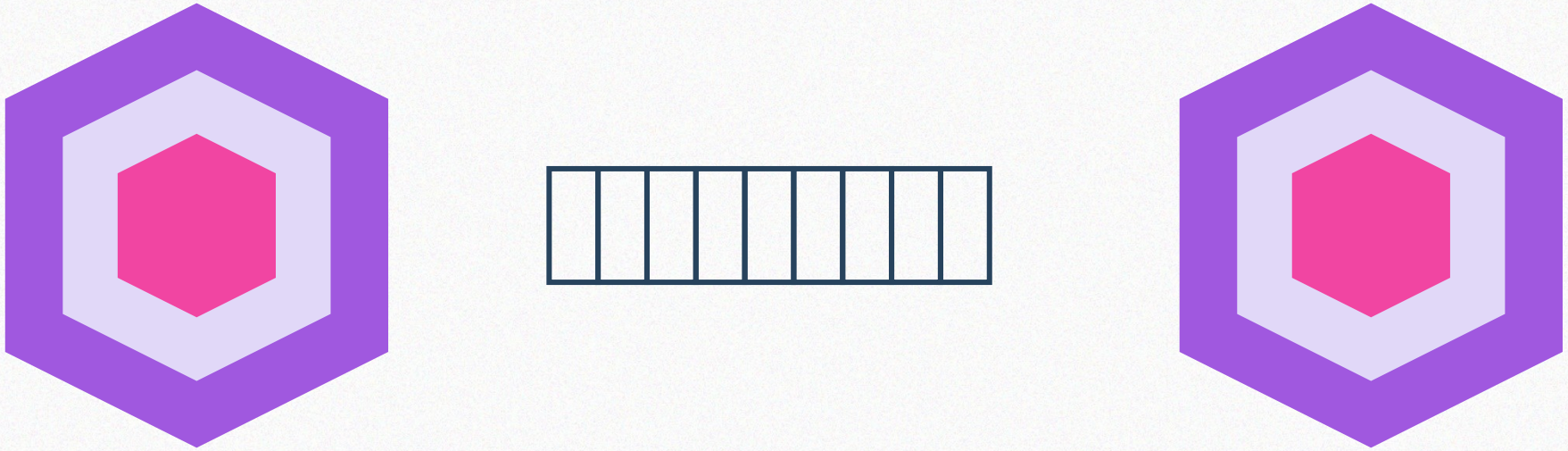
Comunicación Basada en Mensajes

Utiliza un intermediario (broker) para enviar y recibir mensajes. Ejemplos incluyen RabbitMQ y Apache Kafka.

Eventos

Los microservicios publican eventos y otros se suscriben a ellos, permitiendo una actualización en tiempo real

Comunicación Basada en Mensajes



RabbitMQ, SQS, o Kafka

Comunicación Basada en Eventos

Kafka, Google Pub/Sub, AWS SNS.

Comunicación Basada en Eventos

```python
import pika

# Configuración del emisor
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()
channel.queue_declare(queue='task_queue')

# Publicar mensaje
channel.basic_publish(exchange='',
                      routing_key='task_queue',
                      body='Nuevo pedido procesado')

print(" [x] Mensaje enviado")
connection.close()
```

Comunicación Basada en Eventos

```
def callback(ch, method, properties, body):
    print(f" [x] Recibido {body}")

channel.basic_consume(queue='task_queue', on_message_callback=callback,
auto_ack=True)

print(' [*] Esperando mensajes.')
channel.start_consuming()
```